Resolution of Linear Entity and Path Geometries Expressed via Partially-Geospatial Natural Language

by

John Javier Marrero

S.B., C.S. M.I.T., 2008

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 2010

Author_____

Department of Electrical Engineering and Computer Science

February 8, 2010

Certified by_____

Dr. Michael Cleary

Charles Stark Draper Laboratory

Thesis Supervisor

Certified by_____

Dr. Boris Katz

M.I.T. Thesis Advisor

Accepted by_____

Dr. Christopher J. Terman

Chairman, Department Committee on Graduate Theses

Resolution of Linear Entity and Path Geometries Expressed via

Partially-Geospatial Natural Language

by

John Javier Marrero

Submitted to the Department of Electrical Engineering and Computer Science

February 8, 2010

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

When conveying geospatial information via natural language, people typically combine implicit, commonsense knowledge with explicitly-stated information. Usually, much of this is contextual and relies on establishing locations by relating them to other locations mentioned earlier in the conversation. Because people and objects move through the world, a common and useful kind of geospatial phrase is the path expression, which is formed by designating multiple locations as landmarks on the path and relating those landmarks to one another in sequence. These phrases often include nongeospatial information, and the paths often include linear entities. This thesis builds upon the work done for the GeoCoder spatial reasoning system, by addressing several of its limitations and extending its functionality.

Technical Supervisor: Michael Cleary

Title: Decision Systems Group Leader, Charles Stark Draper Laboratory

Thesis Advisor: Boris Katz

Title: Principal Research Scientist

Acknowledgements

I would like to thank my supervisor, Michael Cleary (Draper), and my advisor, Boris Katz (MIT), for their guidance, understanding, and support in the research and writing of this thesis. I would also like to thank Dan Cocuzzo (Draper) and Erik Antelman (Draper) for their help, especially in the early stages of this thesis. Finally, I would like to thank my friends and family for their unending love and support.

This thesis was prepared at the Charles Stark Draper Laboratory, Inc., under the GeoCoder Internal Research and Development Project 21799-001.

_____

John J. Marrero

Table of Contents

Index of Tables

Index of Figures

Chapter 1

Introduction

Natural language is inherently ambiguous. In particular, geospatial expressions used to describe locations often rely on contextual information and common sense knowledge to make sense. For instance, the phrase "the school in Boston" may make sense to someone who knows what and where "Boston" is. The listener, however, must possess this knowledge as well as be able to distinguish which "Boston" is being talked about, as there are multiple locations in the world named "Boston". In addition, the listener must be able to reason about the location and identity of "the school" based on its relation to "Boston". In this case, "the school" is "in Boston", and without any further information, this leads to many possible results. Therefore, the precise meaning of a given geospatial expression is quite difficult to both convey and ascertain, especially if little is known beforehand about the locations being referenced. However, the goal of automated spatial language processing systems is to translate a vague expression such as this to an actual location.

Slagle presents a solution to the problem of geospatial phrase grounding and disambiguation [3]. In her approach, all locations considered in the input phrase have geometries

that can be related to one another via prepositional phrases. That is, instead of defining a location by a single coordinate, she defines a location by a *set* of coordinates which describes its geometry. This allows locations to occupy area and thereby "interact" with one another; for instance, a location may overlap or be contained by another location. Slagle built the GeoCoder spatial reasoning system to realize this approach.

This thesis builds upon Slagle's work (and GeoCoder system) by addressing various limitations and extending the concepts presented therein. Chapter 2 discusses background information and work related to the main components of the system. Chapter 3 covers the main concepts introduced by this thesis as additions or extensions to Slagle's work. Chapter 4 describes the design of the GeoCoder server, its components, and the processes it uses to resolve input phrases passed to it. Chapter 5 gives representative test cases and details the procedures used by GeoCoder when presented with these examples. Chapter 6 discusses open research problems and possible future extensions that could further improve the system. Chapter 7 summarizes the major contributions of this thesis.

Chapter 2

Background and Related Work

This chapter describes other research related to this thesis and is divided into Background and Related Work. Because this thesis builds upon the work done by Slagle, much of the background and related work described therein forms a basis for the discussion in this chapter.

2.1 Background

This section discusses background information relevant to the problems handled by the main aspects of this thesis and the GeoCoder system.

2.1.1 Grounding and Disambiguating Locations

The conversion of a location name to a set of latitude and longitude coordinates by

searching through a database is referred to herein as *grounding*. (The entire process involved in converting an input phrase to this final geometry, including tagging, parsing, reasoning, and grounding is referred to as *resolution.*) This set of coordinates may contain just one pair, designating a point for the location, or a series of coordinates that define the location's shape, designating a geometry for the location. A gazetteer such as GeoNames [15] may be used to translate a name to a point, or a database such as one consisting of the United States Census Bureau's TIGER files [12] may be used to translate a name to a geometry. However, because a particular location name may have many possible results, disambiguation may be needed. Because all results may be equally valid (in particular, GeoCoder considers all results valid), it may not be useful to discard any of them but rather sort them in order of likeliness. This sorting could be based on some heuristic, such as by largest population or area. Leidner gives fifteen common disambiguation heuristics that can be used for this purpose [16].

2.1.2 Reasoning

A description logic [17] is a formal language for knowledge representation that can be used for formal reasoning. A description logic provides a model for *classes* (the terminological knowledge of an application domain), *properties* (the relations between the classes), and *individuals* (the instantiations of the classes with property assertions). These terms are synonymous with *concepts, roles,* and *objects,* respectively.

A description logic's knowledge base is made up of *TBox* (terminological box) and *ABox* (assertional box) statements. The TBox statements describe the hierarchies between the concepts. For instance, the statements "a *cat* is an *animal*" and "a *dog* is an *animal*" describe subclasses of the *animal* class, but "*cat* and *dog* are mutually disjoint" keeps a *cat* from being a *dog,* making them two separate *animal*s . The ABox statements describe where in the hierarchy individuals belong, relative to other individuals and the concepts. For example, "*Ginger* is a *cat*" identifies the individual *Ginger* as being a *cat* (and therefore also an *animal* but never a *dog*).

2.1.3 Path Resolution

In a graph defined as a set of nodes joined by edges, a path is a subset of those edges that leads from a starting node to an ending node. The edges are connected and followed in a specific order. The shortest path problem involves finding such a path where the length is minimized. There are many well-known algorithms for solving this problem, such as A* [17] and Dijkstra's algorithm [18]. However, these algorithms act upon graphs that use points as nodes. This thesis is concerned with finding the shortest path in graphs involving geometries. Furthermore, given the nature of the problem GeoCoder was designed to handle, the actual location of each component of the path is not known beforehand (for instance, it is not known *a priori* where or what "Boston" actually is until it is resolved to a set of coordinates). Each component of the path must be resolved to geometries in order to then find the shortest path through those geometries. This

thesis refers to this problem as *path resolution.*

2.2 Related Work

This section describes systems that have goals similar to those of this thesis, some of which use approaches that this thesis attempts to improve upon.

2.2.1 GeoLogica

GeoLogica [19] automatically interprets geospatial questions by using its reasoner, SNARK, which is unusual in its ability to query external knowledge sources. When SNARK tests an axiom containing a special symbol that represents a knowledge source, a query to the corresponding source is made. GeoCoder uses a similar approach but separates queries to external sources from the reasoner, allowing complete control over what queries are made and when. This thesis describes a modularization of GeoCoder that allows for solutions to problems like path resolution and nongeospatial phrase resolution. This requires specific knowledge source queries to be constructed at certain times, which would not be possible if the reasoner had full control over the queries.

2.2.2 Geospatial Resolution and Pathfinding Systems

There exist many applications for geospatial resolution, such as Google Maps [1] or MapQuest [2]. Such applications can map addresses or names to coordinates, or they can take starting and ending points and find the shortest path between them. For most people, this approach is sufficient, but there are several observable limitations to it. Firstly, the approach relies on the user actually knowing the names corresponding to the path's endpoints, which may not always be the case. Secondly, the approach simplifies locations to points and does not observe the particular geometries or properties that are inherent to any location. Finally, the approach allows no description of the path—while a user may be able to alter the intermediate points of a path and thereby specify the trajectory the path should take, there is no way to describe these points as a person might naturally describe them in words, such as by "the park on Massachusetts Avenue". In contrast, this thesis builds upon the geometric approach to phrase resolution that Slagle developed and integrates it with pathfinding functionality and so avoids such limitations.

2.2.3 START Natural Language Question Answering System

The START natural language question answering system [13] is able to answer questions of both geospatial and nongeospatial nature. For example, a user may ask "what is the capital of

France?" which is a question that relies on a nongeospatial property, "the capital of France", and yields a geospatial answer, "Paris". START seeks out the information it needs to answer questions by accessing a knowledge base that directs it to appropriate knowledge bases on the World Wide Web. GeoCoder uses a similar approach, accessing a knowledge base consisting of TIGER, GeoNames, and any other databases or data collections that may be added in the future. While GeoCoder and START, in essence, share the same goal of returning information based on an input phrase, GeoCoder focuses on input phrases that are geospatial in nature and uses reasoning to arrive at geometric results. START, on the other hand, handles either geospatial or nongeospatial input phrases and returns textual information. This thesis describes functionality for GeoCoder that works with START to handle a greater variety of input phrases that are nongeospatial but have geospatial results, thereby allowing input phrases that are more natural.

Chapter 3

Concepts

This chapter discusses the main concepts introduced by this thesis as additions or extensions to Slagle's original work [3]. Only the ideas are covered here; the specific implementations and usages are explained in the System Design and Solutions chapter.

3.1 Linear Entities

Perhaps the most fundamental extension this thesis makes to Slagle's work is the addition of *linear entities.* Previously, only geometries for areas, referred to as *areal geometries*, were possible (for example, the geometry corresponding to the input phrase "the south of Boston"). An areal geometry is defined by a set of coordinates that describes its boundary. However, this representation makes somewhat less sense for certain concepts, such as for armistice lines or administrative boundaries. Because these concepts have location and therefore have a geospatial presence, they require a geometric representation, but as they do not occupy area, it would be

awkward to represent them with geometries of an areal nature. These are *linear* entities and, in order to represent them, it is necessary to introduce *linear* geometries.

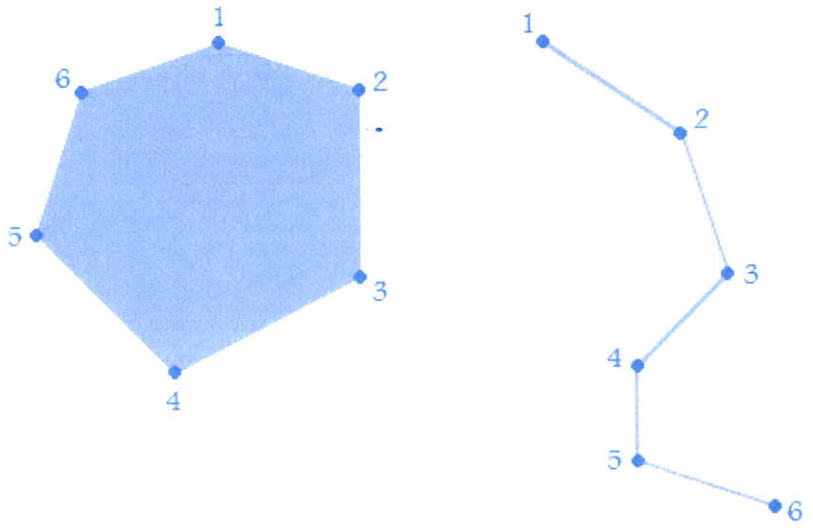

Figure 1: Example of an areal geometry (left) and of a linear geometry (right). The vertices of a geometry are labeled in the order of their definition.

A linear geometry, like an areal geometry, is defined by a set of coordinates. However, for an areal geometry, the coordinates define the *border;* each point is connected to the following point by a line and the final point is connected to the first point. The geometry is then this border and the area contained within the border. For a linear geometry, the set of ordered coordinates is the complete definition of the geometry itself; each point is connected to the following point, but there is no connection from the final point to the first point. Two geometries, one areal and one

linear, are shown in Figure 1.

This thesis adds the linear entity as another possible type of result. That is, linear objects (such as streets and borders, for instance) may now be manipulated and reasoned upon by GeoCoder. The exact means by which areal and linear geometries differ in representation within GeoCoder is described in the System Design and Solutions chapter.

3.2 Path Resolution

A natural extension from the resolution of linear entities is the resolution of paths. This thesis views a path as a sequence of geometries that connect from one location to another. A path in terms of geometries is somewhat different from the typical understanding of a path in terms of points. A path built from points has some indication as to how the points are connected in order; usually these connections are lines. That is, a point connects to another point, which connects to a third point. A linear geometry, then, can be viewed as a path built from points. To traverse a path built from points, the lines connecting those points are followed. When one point in the path is reached, the next connecting line is taken, which leads to the next point in the path. For a path built from geometries, however, any intersecting geometries, whether linear or areal, are considered connected. This means that, if a geometry intersects with four other geometries, then the path can potentially continue into any one of those four geometries as the next geometry. This

is the important, distinguishing subtlety. Therefore, to traverse a path built from geometries, a geometry in the path is followed in its entirety up to the point of intersection with another geometry, whereupon that geometry is then followed in its entirety until the next point of intersection, and so on. This thesis focuses only on paths built from geometries.

Examples of the two kinds of paths are shown below in Figure 2. The figure shows two copies of the same three linear geometries: the first geometry is a straight horizontal line, the second is a straight vertical line, and the third is an "L" shape. The thick circles indicate the points that define the geometries. The thick line denotes the path under discussion. The linear geometries can be thought of as being three streets that cross one another at certain points. The point path on the left is defined by the same points that make up the L-shape geometry. A point path can only be defined by points, so it follows the L-shape exactly and the other two geometries do not define it in any way, despite the fact that the other geometries intersect it. That is, the point path follows just one street and cannot enter the other two streets. The geometric path on the right, however, is not defined by points, but by the geometries it runs through. Therefore, this path can cross over into another geometry if that geometry has intersected with the path's current geometry. This particular path runs through all three streets.

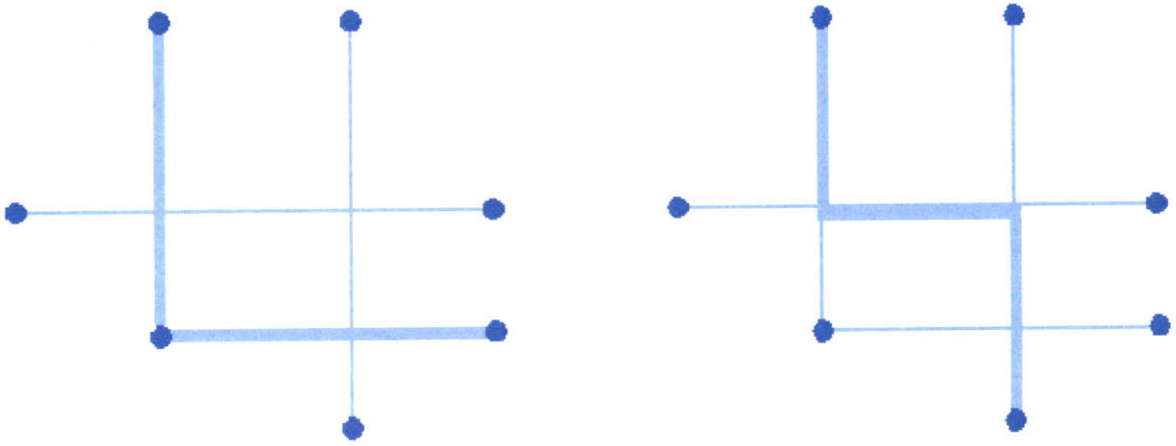

Figure 2: Examples of a point path (left) and a geometric path (right).

When the user gives to GeoCoder an input phrase that is expressing a path, the vertices of the path are the various locations in the phrase that define the course of the path. For example, in the path expression "the path from Symphony Hall to MIT to Harvard", the vertices of the path are "Symphony Hall", "MIT", and "Harvard". The path vertices are joined together by intermediate geometries, which are the aforementioned "intersecting geometries" and, in this example, are the streets that join the locations. These streets may be "Massachusetts Ave", "Prospect St", and so on. (More generally, these intermediate geometries are the *transportation channels* between the locations and can run through land, water, or air.) In the geometric path in Figure 2, the path vertices may be the two circles at the endpoints of the path, while the intermediate geometries are the linear geometries out of which the path is formed.

The actual process of path resolution is discussed in detail in the System Design and Solutions chapter.

3.3 Canonical Phrase Forms

Because natural language allows for many different ways to express the same idea, there are many possible input phrases that may be intended to resolve to the same location. Furthermore, phrases may be arbitrarily complicated or ambiguous. This means that the possible phrase space for a single location is quite large. In order to reduce the space and make the problem more tractable, this thesis makes several constraints on the possible phrases by requiring that the phrases be in *canonical forms.* These canonical forms are predefined structures that the input phrase must take on to be interpretable by GeoCoder. In other words, it is a grammar for the input to GeoCoder. If the input phrase does not fit a canonical form, then it is not guaranteed that GeoCoder will understand the query and thereby produce a useful result. This thesis uses two main canonical forms and their variations (described below in Sections 3.3.1 and 3.3.2), but discusses another possible form in the Future Work chapter.

3.3.1 Prepositional Relation

The prepositional relation form is the usual form that an input phrase takes. This is the form that Slagle's system was built around, and phrases in this form must look like

"*UnnamedFeatureSet* preposition *NamedFeatureSet*". (This form can be viewed as a production rule in a grammar for the input phrases.) A *Feature* is simply a location and a *FeatureSet* is a collection of these *Feature*s, containing only one or multiple *Feature*s related together, possibly by a subphrase or conjunction. An example is "the school in Boston", where "the school" is the only *UnnamedFeature* in the *UnnamedFeatureSet* and "Boston" is the only *NamedFeature* in the *FeatureSet*. The preposition "in" relates the two *FeatureSet*s. According to Slagle, this phrase is a binary prepositional relation.

Another example is "the city between Somerville and Boston", where "the city" is the only *UnnamedFeature* in the *UnnamedFeatureSet* and "Somerville" and "Boston" are the *NamedFeature*s in the *NamedFeatureSet* and are related together by a conjunction. The preposition "between" relates the two *FeatureSet*s. According to Slagle, this phrase is a ternary prepositional relation.

3.3.2 Path Expression

The path expression form is used to identify an input phrase as a path. In other words, it directs GeoCoder to handle the phrase differently than a phrase in the prepositional relation form. Phrases in this form must look like "the path from *vertex* to *vertex* (to *vertex*)*", where *vertex* is a location that acts as a vertex for the path. The component "(to *vertex*)*" indicates that component

of the phrase may be added at the end as many times as desired for each new vertex. Therefore, the first *vertex* (the "from" *vertex*) is the start of the path and each subsequent vertex (each "to" *vertex*) is the next path vertex, in order, until there are no more vertices. (This form, like the prepositional relation form, can be viewed as a production rule in a grammar for the input phrases.) An example of this phrase is "the path from Boston Univ to Boston Common", which designates a path with two vertices, "Boston Univ" and "Boston Common". A vertex may also be defined by a subphrase, such as in "the path from the school on Commonwealth Ave to Boston Common". This phrase form is used to trigger the new path resolution functionality, which is described in the System Design and Solutions chapter.

3.4 Nongeospatial Subphrases

Some query phrases refer to geospatial entities or locations via nongeospatial descriptions. These types of phrases are therefore nongeospatial in nature but may be resolved to equivalent geospatial forms. An example of such a phrase is "the most populous city in Massachusetts", which can be equivalently expressed as "Boston". While the subphrase "in Massachusetts" is geospatial, the subphrase "the most populous city" is not, and so the phrase as a whole is nongeospatial. This is a phrase that a human being would have no trouble interpreting geospatially, assuming they actually know which city in Massachusetts is the most populous.

Slagle's original system can only understand "in Massachusetts" (although the "in" does not provide any additional information). If the nongeospatial subphrase were resolved to a geospatial one, then the entire phrase would be geospatial and therefore resolvable to a geometry. The System Design and Solutions chapter describes an extension to Slagle's system that allows the resolution of nongeospatial subphrases to geospatial ones, by using an external system.

3.5 Scale

It is difficult to define what it means for two locations to be "near" one another. Logically, it may seem that "nearness" to a particular location is a gradient that decreases with distance from that location. However, computer systems typically demand absolutes, and those are what Slagle worked with. Therefore, this thesis follows the paradigm that, if a location X is within a certain distance threshold to location Y, then X is "near" Y. Otherwise, it is "not near". For this to be reasonable, however, a variable threshold must be used, as the distance threshold used for determining the "nearness" of two universities may not make sense if used for determining the "nearness" of two cities—that is, saying "MIT is near Harvard" suggests a different scale of "nearness" than saying "Boston is near New York City". This thesis uses the computed area of a location to determine another location's "nearness" to it, allowing for similar input phrases that use "nearness" relationships to produce different behavior, depending on the sizes of the

locations involved. The exact mechanism is described in the System Design and Solutions chapter.

Chapter 4

System Design and Solutions

This chapter describes the GeoCoder system and how its components relate and work together to solve the various aforementioned problems.

4.1 Modularization

The solution presented by Slagle, for the most part, consists of a single monolithic algorithm which builds the ontology model, accesses the databases, grounds the features, converts the geometries to an XML output string, and displays the results (see Appendix A.1). Essentially all of the work is handled by one module, except when third-party tools such as the Stanford Parser and the Pellet Reasoner are invoked. This approach is sufficient for solving a single problem that may have some variations. This work found it valuable to restructure and decompose the algorithm into submodules, as this offers flexibility in how they may be used and how they may interact with one another. More specifically, this produces a *data path*, over which

data is manipulated by the various modules in some combination and the output of one module is passed as the input to the next module. So the data path starts with an input phrase which is processed through a particular module, then the result of that module is processed through another module, and so on, until the final result is achieved. This also allows modules to be reused, as results may be fed back into previously-used modules for further processing. Furthermore, at each step, *any* of the results up to that point may be used as the input for the next step; a module is not required to use only the output from the previous module. Now it is the specific problem at hand that dictates what modules are used and in what combination.

Under this paradigm, the algorithm written by Slagle may be broken into five major modules, each of which has tools and subalgorithms associated with it:

- *Tagger* – performs named-entity tagging on a phrase (invokes the Stanford NE Tagger [4])
- *Parser* – parses a phrase and produces its parse tree (invokes the Stanford Parser [5] and Slagle's Parse Tree Translation Algorithm)
- *Reasoner* – maintains an ontology model and applies rules (invokes the Jena Rule Reasoner [6] and Pellet Reasoner [9])
- *Grounder* – resolves a phrase to a corresponding geometry (invokes PostGIS [10] and the Java Topology Suite [11], uses TIGER files [12] and GeoNames [15] as databases)

- *Displayer* – displays geometries (passes results to Google Earth [14])

The first four modules (the Tagger, Parser, Reasoner, and Grounder modules) may be chained together and abstracted as a new module called the *Resolver*. This is almost exactly Slagle's same algorithm, which takes in an input phrase and returns the corresponding geometries, but now it will be used in a larger context and for certain kinds of phrases, and the result is not displayed unless passed to the Displayer module. This thesis uses the five modules and the Resolver in various combinations, in addition to the *Pathfinder* and *Arbiter* modules that will be introduced in Sections 4.2 and 4.3.2, respectively.

4.2 Path Resolution

The problem of path resolution is divided into these three smaller problems:

- Resolution of the path vertex geometries (locations on the path)
- Resolution of all connecting, intermediate geometries (transportation channels)
- Pathfinding on geometries

The Resolver module provides the answers for the first two subproblems. The difference between the two subproblems is in what is asked of the Resolver.

The first subproblem is the resolution of the path vertex geometries. The path vertices are the various locations that define the path. These vertices are given in the path expression and so their names are known. For example, in the path expression "the path from Symphony Hall to MIT to Harvard", the vertices of the path are "Symphony Hall", "MIT", and "Harvard". The names of these vertices can be passed to the Resolver and thereby converted to geometries.

The second subproblem is the resolution of *all* such geometries. Therefore, if there are many possible ways to connect any two sequential path vertices, then all geometries that belong to *any* of these paths are included in the result set of intermediate geometries. The input query to the Resolver depends on the possible transportation channels between the path vertices. These transportation channels may be streets, subways, trains, waterways, and so on. For example, "the path from Boston University to Boston Common" has two vertices and many streets between these two vertices. For simplicity, this example ignores the possibility of other transportation channels between these two locations. Passing in the query "the streets between Boston University and Boston Common" to the Resolver will return the intermediate geometries in the area between the two locations. Of course, there may be many other possible paths that involve streets outside this area between the two locations, but this thesis considers only this reduction of the possibility space, for the purposes of practicality.

Once the geometries for the path vertices and all connecting transportation channels have

been resolved, only the subproblem of pathfinding on these geometries remains, using the start and destination locations as the endpoints of the path. This thesis solves this problem by using a modified form of Dijkstra's algorithm, which acts on geometries rather than on points (see Appendix A.2). The resulting set of geometries is the shortest path between the path vertices. This example contains only two path vertices, but if there were more then this process would be repeated to find the subpath between each pair of sequentially-adjacent vertices, and the final result would be the union of all the subpaths.

This entire process may be abstracted as the *Pathfinder* module, which returns the set of geometries for a path given a path expression.

4.3 Nongeospatial Phrase Resolution

This section details an extension to Slagle's system that allows the resolution of nongeospatial subphrases to geospatial ones through the accessing of a knowledge base. For this purpose, this thesis uses the START natural language question answering system developed by Katz [13], which is able to return answers for questions that use the nongeospatial properties of locations, among other things.

In order for GeoCoder to be able to ask questions of START, an interface was developed by Katz, by which GeoCoder can send an HTTP request containing a phrase to START and

receive an XML string from START containing the response. GeoCoder can then parse this XML string to extract the relevant results and information about the results, as necessary. (For example, sending an HTTP request with the phrase "capital of France" returns an XML string containing "Paris" along with information that it is the capital of the country France.) If START cannot find a result for the query, the XML string states "DONT-KNOW". If the response contains a list of results and the query phrase mentioned a property by which the results can be compared, then the list is sorted according to that property. For example, if the query phrase is "the most populous cities in Massachusetts", then the response will contain a list of these cities sorted by population.

Using this interface, GeoCoder can pass to START nongeospatial questions that have geospatial answers and can thereby handle previously-uninterpretable queries, as long as START is capable of providing answers to GeoCoder's questions.

4.3.1 Phrase Partitioning

Before GeoCoder can ask questions of START, it needs to decide *what* questions to ask. Beginning from the complete input phrase, GeoCoder must select appropriate subphrases to be passed to START. These subphrases should be nongeospatial and would thereby be converted to equivalent geospatial subphrases. In order to select these subphrases, the complete input phrase must be partitioned in some fashion. However, due to the variability of all possible input phrases,

the structure of any given phrase may be uninterpretable under any single approach. This thesis details two partitioning approaches and combines them to increase the precision of the final results.

4.3.1.1 Recursive Phrase Tree

The first approach partitions the input phrase using a tree structure. This tree, referred to herein as the *phrase tree,* is derived from the parse tree that GeoCoder selects for the complete input phrase. The parse tree is generated by the Parser module. The parse tree is used to generate the phrase tree because the parse tree provides information about the structure of the phrase and therefore provides information as to how the components of the phrase are related to one another (for instance, via prepositions). This knowledge of component relationships can be used to guide the phrase partitioning. The transformation of a parse tree to a phrase tree proceeds as follows:

1) All tagging is removed from the parse tree nodes. These tags may have been placed by the Tagger module. For example, a node may be labeled "Massachusetts/LOCATION", indicating that the word "Massachusetts" refers to a location. After this step, the label would simply read "Massachusetts".
2) Any interior node with a single child is deleted and the child is moved up in the tree to

replace its parent. This step is repeated until there are no more interior nodes with one child.

3) Starting from the leaves and moving upwards level by level, the label for each node in the tree is replaced with the concatenation of all the labels of the children nodes (in order from left to right).

As an example, the phrase "the most populous city in Massachusetts" will yield the parse tree depicted below in Figure 3 when passed through the Parser.

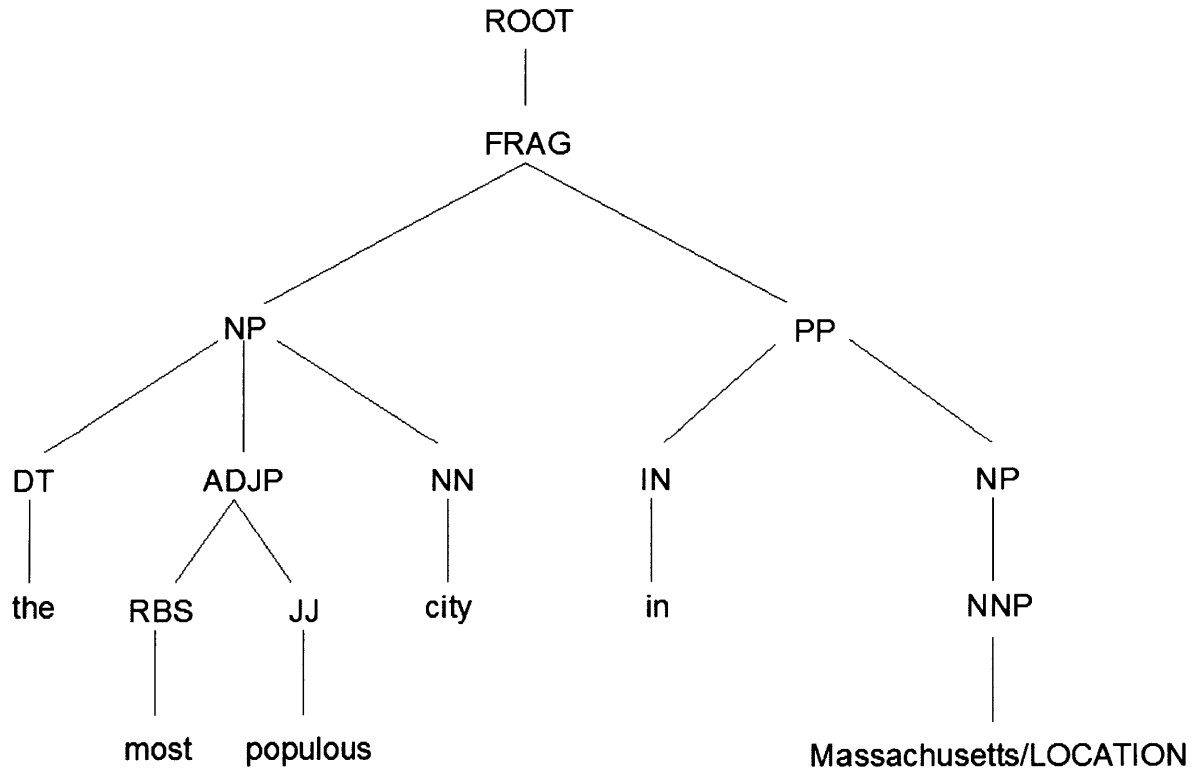

Figure 3: Parse tree for the phrase "the most populous city in Massachusetts".

Step 1 of the transformation algorithm removes the "/LOCATION" tag from the node labeled with "Massachusetts/LOCATION", leaving the label "Massachusetts". Step 2, after several iterations, removes the "DT", "RBS", "JJ", "NN", "IN", "NP", "NNP", and "ROOT" nodes. Step 3 then sets the labels for each node to the concatenation of its children's labels, starting from the bottom level and moving up. Thus, the "ADJP" node, which is the parent for the "most" and "populous" nodes, takes on the label "most populous", and its parent, the "NP" node, takes on the label "the most populous city". Similarly, the "PP" node takes on the label "in Massachusetts" and the "FRAG" node takes on the label "the most populous city in Massachusetts". The resulting phrase tree is shown below in Figure 4.

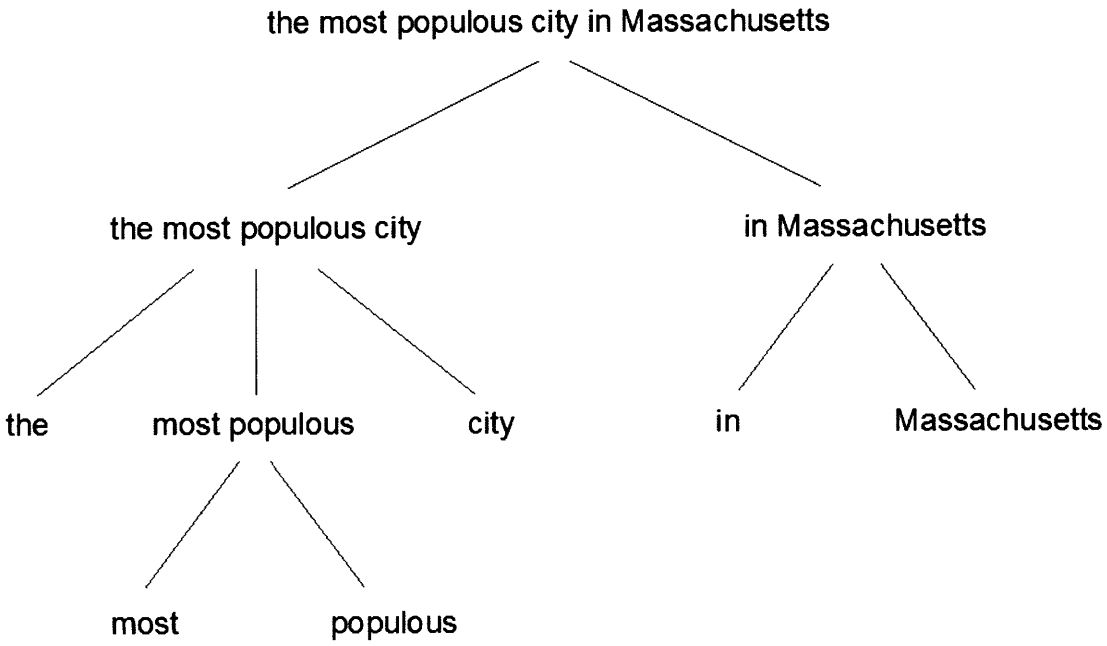

Figure 4: Phrase tree for the phrase "the most populous city in Massachusetts".

Once the phrase tree has been generated from the parse tree, the following algorithm is executed:

1. Begin at the right-most leaf. Set the label for this node as the current subphrase.
2. Query the Resolver module for a result for the current subphrase. If the Resolver:
   (a) has a result, mark this node (and therefore its subphrase) as "successful".
   (b) cannot find a result (perhaps because it does not understand the subphrase), ask START for a result. If START:

i. finds an answer, mark the node as "successful", replace the node's label with the new answer from START, and propagate the change upwards to all ancestor nodes' labels. Repeat step 2 on the new node.

ii. does not find an answer, mark the node as "failed".

3. If the last subphrase:

(a) was successful, and:

i. there are two or more remaining siblings to the left, move to the next-left sibling and set its label as the current subphrase. Repeat step 2 for this node.

ii. there are no more siblings or only one remaining, and:

A. there is a parent, move to the parent, set the current subphrase to the parent's label, and repeat step 2 on the parent.

B. there is no parent, this was the root node and the entire input phrase has succeeded, so terminate.

(b) was a failure, and:

i. there is a parent, move to the parent and set its label as the current subphrase. Repeat step 2 for this node.

ii. there is no parent, this was the root node and the entire input phrase has failed, so terminate.

The algorithm attempts to resolve subphrases using GeoCoder's Resolver module before falling back to START. The rationale behind this behavior is that if a subphrase can be resolved without START, then the subphrase is geospatial (an example is "the school in Boston"); otherwise the subphrase is perhaps nongeospatial (it may also simply be a geospatial phrase that GeoCoder's Resolver module does not understand) so START may be used to attempt to get a result for it and then resolve it to a geospatial subphrase, whereupon it may then be resolved to a geometry by the Resolver (an example is "the capital of France", which is resolved to "Paris", which can then be resolved to a geometry).

Step 3.a checks for the presence of more or two siblings to the left of the current node. The final, leftmost sibling is skipped because the algorithm is designed around a right-sided heuristic, where branches are considered more important components to the phrase the more rightward they are. Under this heuristic, the leftmost sibling, usually a determiner or a preposition, is deemed unimportant to the phrase and therefore can be ignored.

4.3.1.2 Leftward Phrase Expansion

The second approach uses only the input phrase and no auxiliary structure. It follows this algorithm:

1. Start with the right-most word and set that as the subphrase.
2. If the current subphrase:
   (a) is the entire input phrase, query the Resolver for a result. Return this result and terminate.
   (b) is not the entire input phrase, query START for a result. If START:
      i. finds an answer, replace the subphrase with the new answer from START (therefore replacing that part of the entire input phrase). Repeat Step 2 on the new subphrase.
      ii. does not find an answer, continue to step 3.
3. Expand the current subphrase leftward by one word.
4. Return to step 2.

4.3.2 Arbiter

The phrase tree generation algorithm and the two aforementioned approaches to phrase partitioning are executed and combined by the *Arbiter* module. This module's task is to determine what subphrases to pass and to where. Namely, subphrases can be passed to GeoCoder's Resolver

module or to START. The Arbiter generates the phrase tree and then executes the two partitioning approaches in order; it first tries the recursive phrase tree approach with the phrase tree and if it is successful then it returns the results. If the first approach yields no results, then the Arbiter tries the leftward phrase expansion approach and returns the results. If no results are found from either approach, then the nongeospatial phrase resolution failed.

4.4 Scale

When searching in TIGER for the geometries that are "near" a particular geometry, GeoCoder attaches the following *withinDistance* string to its PostGIS query:

```
ST_DWithin(the_geom, GeomFromText('ref_geom_wkt',4326),
40*(ST_Area(the_geom) + ST_Area(GeomFromText('ref_geom_wkt',4326))))
and not equals(the_geom,GeomFromText('ref_geom_wkt',4326))
```

In the string, `the_geom` is the current geometry under consideration, whose nearness to the particular geometry `ref_geom_wkt` is being tested. The key aspect to notice is the use of the areas of both of the geometries as part of the query itself. The areas for the two geometries are computed, added, and then multiplied by a scaling factor of 40 (this particular value was chosen empirically and may be adjusted). The resulting value is the threshold used to determine whether two geometries are near one another.

4.5 Dynamic Ruleset

One of the main concerns in building a scalable system is computational performance. In particular, the Reasoner module begins to perform very slowly as the number of the assertion rules it uses increases, but for any complicated reasoning system, a large number of rules is unavoidable. Furthermore, many of these rules are very complex individually; for example, GeoCoder uses many rules that are similar in form to the following:

> [(?a parse:*child* ?b), (?a *rdf:type* parse:*NP*), (?b *rdf:type* parse:*NP*), (?a parse:*child* ?c), (?c *rdf:type* parse:*PP*), (?c parse:*child* ?d), (?c parse:*child* ?e), (?e *rdf:type* parse:*NP*), (?d *rdf:type* domain:*between*), (?b parse:*child* ?g), (?g rdf:*type* parse:*Noun*), (?e parse:*child* ?f), (?f rdf:*type* base:*FeatureSet*) → (?g relation:*between* ?f)]

There is one such rule for every preposition in the English language. As a result, both the number and the size of the rules become problematic. However, not all of these rules are used at all times. Therefore, this thesis uses a dynamic ruleset to limit the number of rules that are active and loaded into the reasoner at any given time.

The process for creating the ruleset is straightforward: GeoCoder scans through the input phrase to determine what prepositions are present (and therefore what rules will potentially be needed to match a corresponding prepositional relation pattern), collects all of the rules relevant to those prepositions, and only loads those rules into the reasoner. This significantly improved performance across all input phrases, reducing the time taken to produce a result approximately thirty-fold..

4.6 System Process

When given an input phrase, GeoCoder follows the process shown in Figure 5. There are two processing branches that the system can take. In order to determine which branch to take, it first analyzes the phrase to determine if the phrase is in the prepositional relation canonical form (described in Section 3.3.1). If so, GeoCoder performs phrase resolution, which first invokes the Tagger and Parser to generate a tagged parse tree that the Arbiter then uses to generate the phrase tree for the input phrase (as described in Section 4.3.1.1). This phrase tree is then used by the Arbiter to perform phrase partitioning using the phrase tree (also described in Section 4.3.1.1). If that approach fails, the Arbiter performs phrase partitioning using leftward phrase expansion (described in Section 4.3.1.2). If either approach is successful, the resulting phrase is passed to the Resolver to attain a geometry, which is then passed to the Displayer for presentation. Note that the Arbiter, while performing phrase partitioning, makes many calls to the Resolver (as described in Sections 4.3.1.1 and 4.3.1.2).

Alternatively, if the input phrase is in the path expression canonical form (described in Section 3.3.2), GeoCoder performs path resolution. Path resolution itself contains a series of three instances of phrase resolution (as the start and end path vertices and the intermediate geometries must be resolved). The entire process of path resolution repeats for additional pairs of vertices, if the path is longer than two vertices. The results are then passed to the Displayer for presentation.

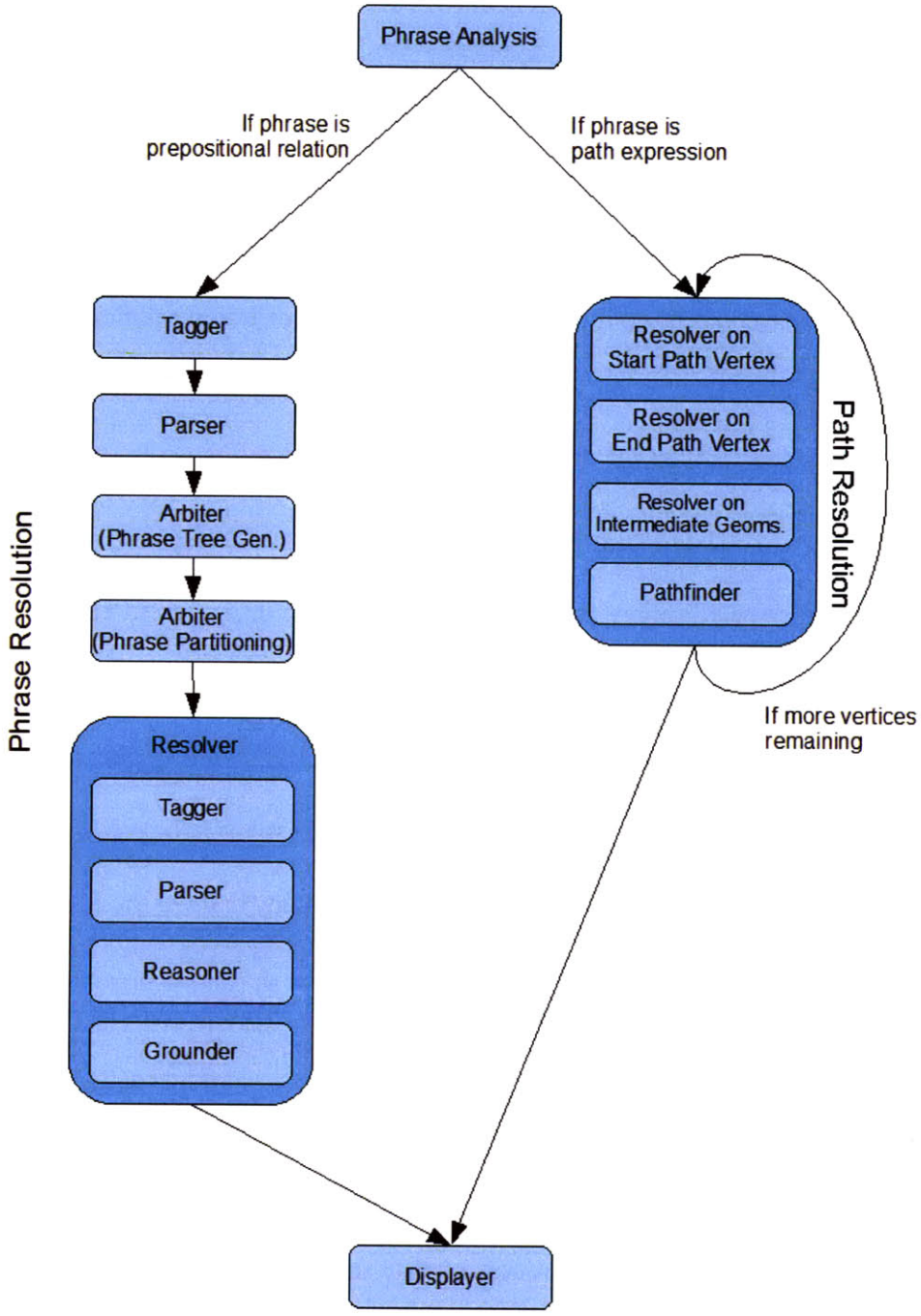

Figure 5: Flow diagram of the system process.

4.7 User Interface

The user interface remains largely unchanged from that built by Slagle. GeoCoder uses a client-server architecture; the server's parameter-setting interface is shown in Figure 6, and the client run on Google Earth, through which the user gives GeoCoder input phrases, is shown in Figure 7.

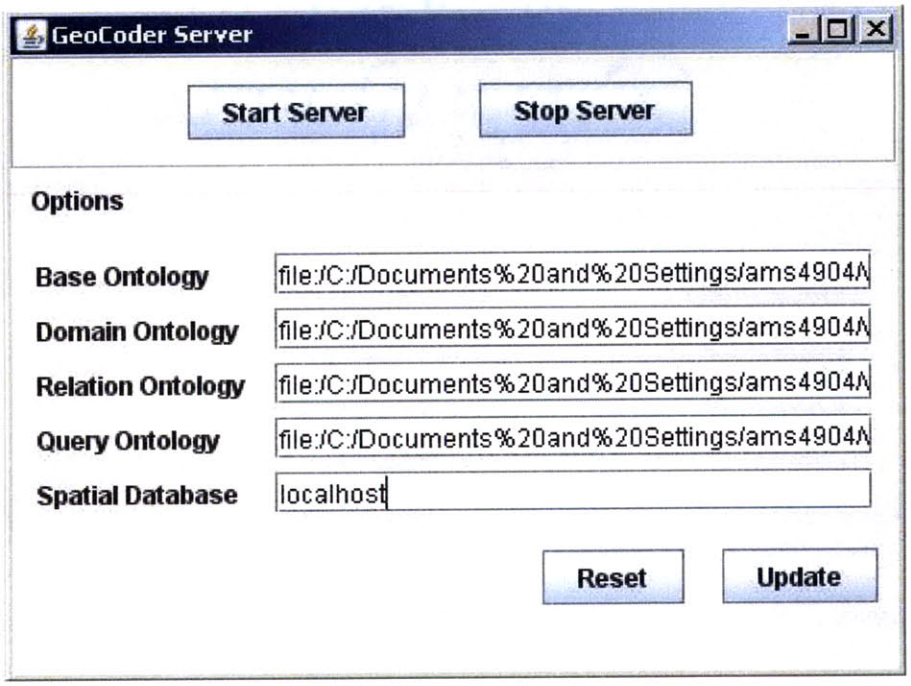

Figure 6: Screenshot of the GeoCoder server. *(Image credit: Slagle [3].)*

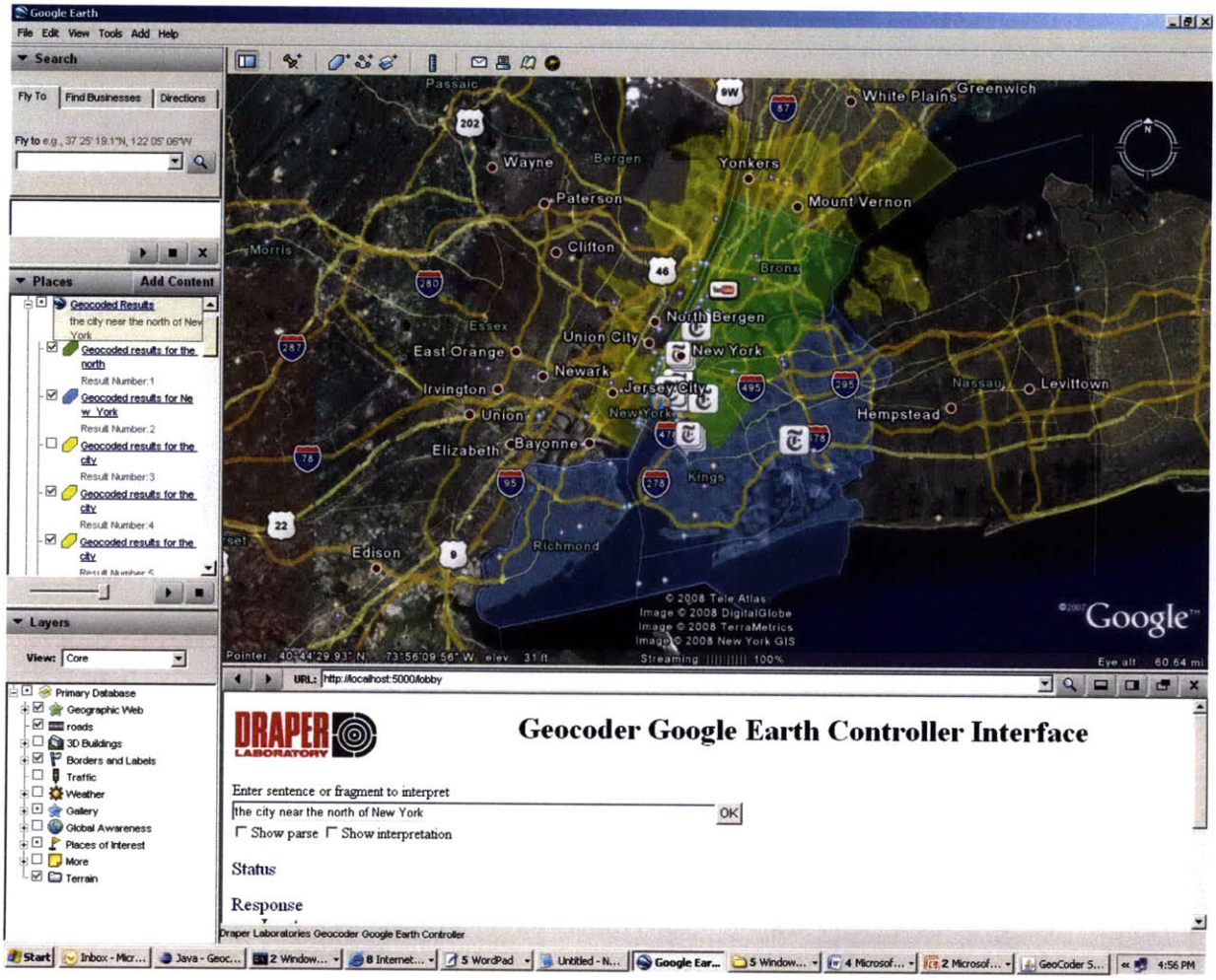

Figure 7: Screenshot of the GeoCoder client. *(Image credit: Slagle [3].)*

4.8 Ontology Structure

The ontology structure from Slagle's original work has been rewritten by our research group to more effectively group items that are logically related and to better accommodate new

additions. The original ontology structure is shown in Figure 8, while the new ontology structure is shown in Figure 9. Arrows indicate dependency while dotted arrows indicate dependency through rules.

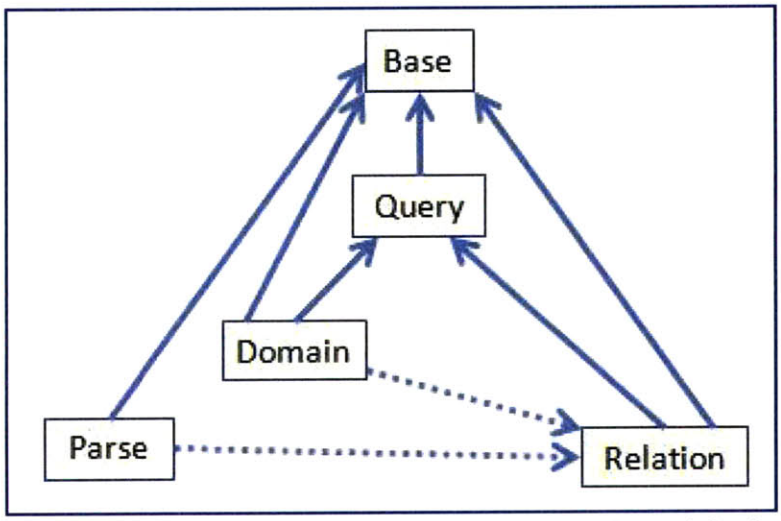

Figure 8: The ontology structure used in Slagle's original GeoCoder system. *(Image credit: Slagle [3].)*

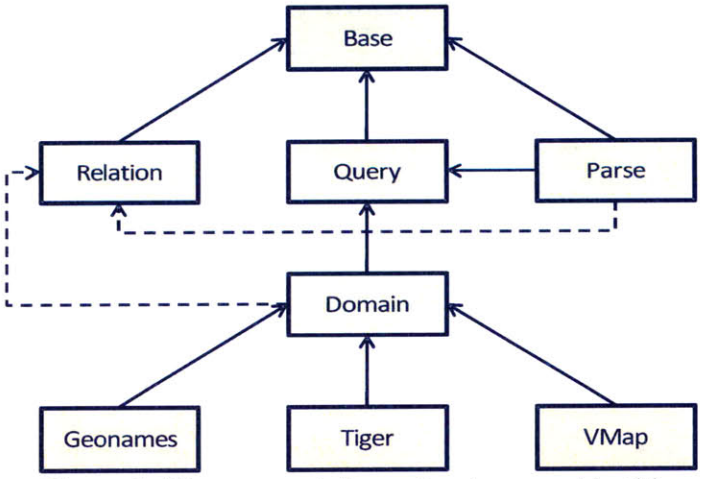

Figure 9: The new ontology structure used in this thesis's GeoCoder system.

Chapter 5

Examples

This chapter uses several examples to demonstrate the procedures that GeoCoder follows. As described in Section 4.1, this thesis has modularized GeoCoder into the Tagger, Parser, Reasoner, Grounder, and Displayer modules, which collectively are the Resolver module. In addition, this thesis introduced the Pathfinder and Arbiter modules. In explaining each example, this section will give the sequence of modules used, the input to each module, and the corresponding output from each module. In the cases where the procedure followed is exactly the same as Slagle's original algorithm, the process will be explained in terms of the five modules comprising the Resolver, as opposed to using the same nine steps as Slagle.

5.1 Simple Linear Entity Grounding and Reasoning

This section describes examples dealing with linear entities and how they relate to areal entities. These examples use input that is in the simplest prepositional relation canonical form

(described in Section 3.3.1, these are the forms Slagle's system could handle) and they do not introduce any complications.

5.1.1 Linear Entities Derived from Areal Entities

The input phrase for this example is "the streets between Boston Univ and Boston Common", where "the streets" are the desired linear entities that will be derived from the areal entities "Boston Univ" and "Boston Common", using the "between" relation. Because this is a simple grounding example, the procedure followed for a phrase is exactly that used in Slagle's algorithm. For simplicity of explanation the Arbiter module will be ignored here and will be explained only in later examples when START is needed. As a result, the procedure followed here is a straightforward chain through all the modules of the Resolver module.

1) Tagger adds tags to input phrase.
2) Parser produces parse tree from tagged phrase.
3) Reasoner asserts rules based on parse tree.
4) Grounder determines geometries by using rule assertions and databases.
5) Displayer presents geometries.

These steps are detailed below and then summarized with their results in Table 1.

Tagger: The GeoCoder server passes the input phrase "the streets between Boston Univ and Boston Common" to the Tagger module. The Tagger module determines that "Boston Univ" and "Boston Common" are locations and tags them as such, resulting in the tagged phrase "the streets between Boston_Univ/LOCATION and Boston_Common/LOCATION".

Parser: The tagged input phrase is passed to the Parser module, which returns the parse tree in Figure 10.

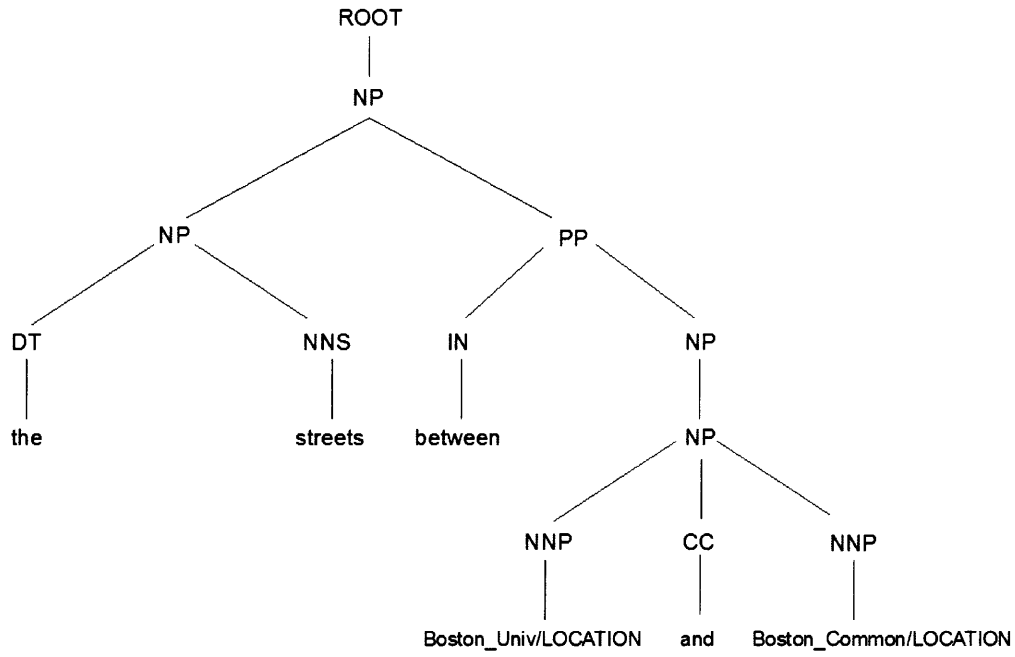

Figure 10: Parse tree for the phrase "the streets between Boston_Univ/LOCATION and Boston_Common/LOCATION".

Reasoner: The parse tree is passed to the Reasoner module, which creates individuals corresponding to each node in the parse tree, adds these individuals to the ontology model, recognizes patterns in the parse tree as dictated by the rules, and asserts the appropriate relationships. (For a more detailed explanation of the rules and their use, see Slagle [3].) The "(NNS streets)" node is given a class assertion of *Street,* while the "(NNP Boston_Univ/LOCATION)" and "(NNP Boston_Common/LOCATION)" nodes are given class assertions of *NamedFeature.* Because *NamedFeature* is a subclass of *Feature,* "(NP (NNP

Boston_Univ/LOCATION))" matches the rule

> [(?a *rdf:type* parse:*NP*), (?a parse:*child* ?b), (?b *rdf:type* base:*Feature*) → (?a *rdf:type* base:*FeatureSet*)]

which causes "(NP (NNP Boston_Univ/LOCATION))" to be asserted as a *FeatureSet,* which then triggers the rule

> [(?a *rdf:type* base:*FeatureSet*), (?a parse:*child* ?b), (?b *rdf:type* base:*Feature*) → (?a base:*hasFeature* ?b)]

and adds "*hasFeature* (NNP Boston_Univ/LOCATION)". Similarly, "(NP (NNP Boston_Common/LOCATION))" and "(NP (DT the) (NNS streets))" become *FeatureSet*s with "*hasFeature* (NNP Boston_Common/LOCATION)" and "*hasFeature* (NNS streets)", respectively.

Now moving up a level, the "(NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION))" individual becomes a *FeatureSet*, with "*hasFeature* (NNP Boston_Univ/LOCATION)" and "*hasFeature* (NNP Boston_Common/LOCATION)". Now the following rule is matched:

> [(?a parse:*child* ?b), (?a *rdf:type* parse:*NP*), (?b *rdf:type* parse:*NP*), (?a parse:*child* ?c), (?c *rdf:type* parse:*PP*), (?c parse:*child* ?d), (?c parse:*child* ?e), (?e *rdf:type* parse:*NP*), (?d *rdf:type* domain:*between*), (?b parse:*child* ?g), (?g rdf:*type* parse:*Noun*), (?e parse:*child* ?f), (?f rdf:*type* base:*FeatureSet*) → (?g relation:*between* ?f)]

and the following relation is asserted:

> (NNS city) *between* (NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION))

establishing a *between* relation on "streets" and "Boston Univ and Boston Common". This leads to the assertion of the *intersects-buffer-convexHull* property, which will be used later to direct the construction of the database query to TIGER.

Grounder: The assertions made by the Reasoner module are now used by the Grounder module. The *NamedFeature*s for "Boston Univ" and "Boston Common" are grounded, yielding a *MultiPolygon* geometry and a *GroundedFeature* assertion for each. (For a more detailed explanation of how the *NamedFeature*s are grounded, see Slagle [3].) The ungrounded *FeatureSet* "(NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION))" is then grounded, yielding a GeometryCollection that contains the geometries for both of the new *GroundedFeature*s.

In order to ground the *UnnamedFeature* for "streets" with the *GroundedFeatureSet*s, an *intersects-buffer-convexHull* PostGIS query to TIGER is built, where ellipses denote long sequences of coordinates that have been omitted for clarity of the example:

```
select asBinary(the_geom) from edges where
(mtfcc = 'S1200' or mtfcc='S1300' or mtfcc='S1400') and
((intersects(the_geom,
   buffer(convexHull(GeomFromText('MULTIPOLYGON(...)',4326)),
   0.1*ST_Distance(GeometryN(GeomFromText('MULTIPOLYGON(...)',4326),1),
                   GeometryN(GeomFromText('MULTIPOLYGON(...)',4326),2))))
   and not within(the_geom,GeomFromText('MULTIPOLYGON(...)',4326))))
```

This computes the convex hull of the two geometries for Boston University and Boston Common and then looks in the *edges* table in the TIGER database for the linear geometries that intersect this hull. (For an overview of the TIGER relationship tables, see Appendix A.3.) There are some size adjustments made to the convex hull which are explained in more detail in Slagle's thesis [3]. The linear geometries that intersect the hull must also have MTFCC code "S1200", "S1300", or "S1400", which are the codes for the various kinds of streets in the TIGER database. Of course, this is a database-specific constraint and the exact requirement may vary depending on the database that is used.

The end result is a set of 2 areal geometries (defined by a *MultiPolygon* for "Boston Univ" and a *MultiPolygon* for "Boston Common") and 1224 linear geometries (defined by *MultiLineStrings* for the streets that are between Boston University and Boston Common).

Displayer: The set of geometries from the Grounder module is converted into a KML string for display in Google Earth:

```
<ENTITY><NAME>streets</NAME><GML>
 <gml:MultiLineString srsName='0'>
  <gml:lineStringMember>
    <gml:LineString srsName='0'>
      <gml:coordinates>
            -71.092135,42.34668 -71.092207,42.346691 -71.092314,42.346694
      </gml:coordinates>
    </gml:LineString>
  </gml:lineStringMember>
 </gml:MultiLineString>
</GML></ENTITY>
<ENTITY><NAME>streets</NAME><GML>
 <gml:MultiLineString srsName='0'>
  <gml:lineStringMember>
    <gml:LineString srsName='0'>
      <gml:coordinates>
            -71.067898,42.348616 -71.06804199999999,42.348776
-71.068123,42.348914
      </gml:coordinates>
    </gml:LineString>
  </gml:lineStringMember>
 </gml:MultiLineString>
</GML></ENTITY>

(one ENTITY entry for each street segment)

<ENTITY><NAME>streets</NAME><GML>
 <gml:MultiLineString srsName='0'>
  <gml:lineStringMember>
    <gml:LineString srsName='0'>
      <gml:coordinates>
            -71.08355,42.352457 -71.083356,42.352042999999995
      </gml:coordinates>
    </gml:LineString>
  </gml:lineStringMember>
 </gml:MultiLineString>
</GML></ENTITY>
```

The resulting display is shown below in Figure 11. For this figure, Google Earth was used as the rendering engine. The satellite imagery comes from Google Earth, while the yellow areal and linear entities marked in the display are the geometries passed by GeoCoder. The text "(Display credit: Google Earth.)" will be used in the captions of the figures throughout this thesis to express this arrangement.

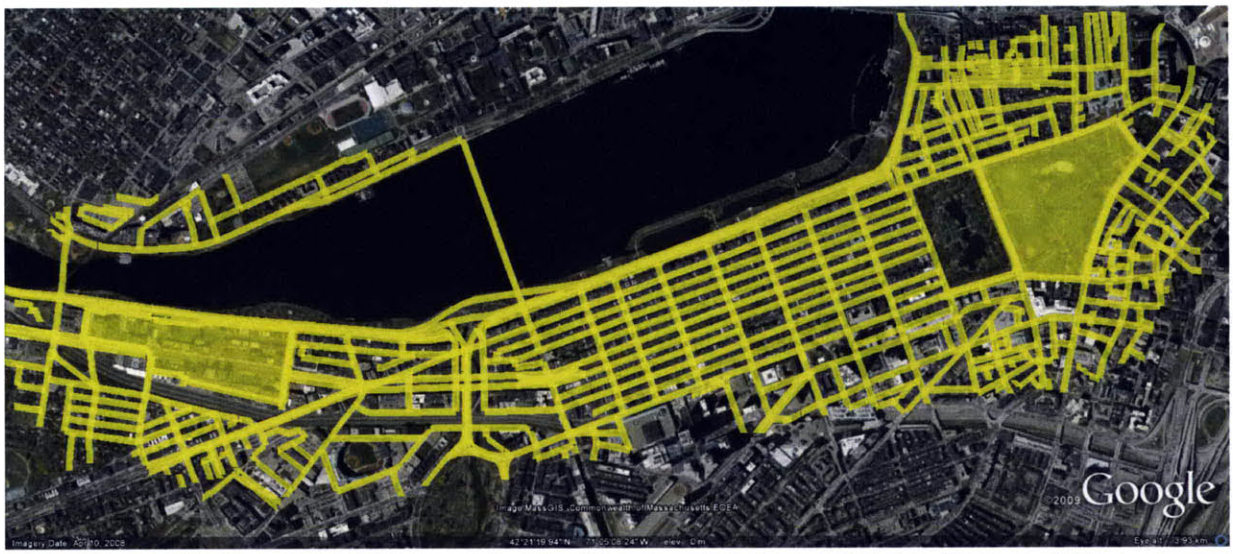

Figure 11: Result set of geometries for "the streets between Boston Univ and Boston Common". *(Display credit: Google Earth.)*

Figures 12 and 13 show zoomed versions of the result set in the areas around the two areal geometries.

Figure 12: Zoomed view of the result shown in Figure 11, in the area around Boston University. *(Display credit: Google Earth.)*

Figure 13: Zoomed view of the result shown in Figure 11, in the area around Boston Common. *(Display credit: Google Earth.)*

Table 1 gives a summary of the processing results in this example.

Module	Result
Tagger	"the streets between Boston_Univ/LOCATION and Boston_Common/LOCATION"
Parser	(ROOT (NP (NP (DT the) (NNS streets)) (PP (IN between) (NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION))))))
Reasoner	(NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION)) *rdf*:type *FeatureSet*; (NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION)) *hasFeature* (NNP Boston_Univ/LOCATION); (NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION)) *hasFeature* (NNP Boston_Common/LOCATION); (NP (DT the) (NNS streets)) *rdf*:type *FeatureSet*; (NP (DT the) (NNS streets)) *hasFeature* (NNS streets); (NNS streets) *between* (NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION)); (NNS streets) *intersect-buffer-convexHull* (NP (NNP Boston_Univ/LOCATION) (CC and) (NNP Boston_Common/LOCATION)); (NNS streets) *rdf*:type *ExistingFeature*;
Grounder	1 *MultiPolygon* for "Boston Univ"

	1 *MultiPolygon* for "Boston Common" 1224 *MultiLineStrings* for "streets"

Displayer	`<ENTITY><NAME>streets</NAME><GML>`
	`<gml:MultiLineString srsName='0'>`
	`<gml:lineStringMember>`
	`<gml:LineString srsName='0'>`
	`<gml:coordinates>`
	`(...)`
	`</gml:coordinates>`
	`</gml:LineString>`
	`</gml:lineStringMember>`
	`</gml:MultiLineString>`
	`</GML></ENTITY>`
	
	`(one ENTITY for each street segment)`
	
	`<ENTITY><NAME>Boston Univ</NAME><GML>`
	`<gml:MultiPolygon srsName='0'>`
	`<gml:polygonMember>`
	`<gml:Polygon srsName='0'>`
	`<gml:outerBoundaryIs>`
	`<gml:LinearRing srsName='0'>`
	`<gml:coordinates>`
	`(...)`
	`</gml:coordinates>`
	`</gml:LinearRing>`
	`</gml:outerBoundaryIs>`
	`</gml:Polygon>`
	`</gml:polygonMember>`
	`</gml:MultiPolygon>`
	`</GML></ENTITY>`
	
	`<ENTITY><NAME>Boston Common</NAME><GML>`
	`<gml:MultiPolygon srsName='0'>`
	`<gml:polygonMember>`
	`<gml:Polygon srsName='0'>`
	`<gml:outerBoundaryIs>`
	`<gml:LinearRing srsName='0'>`
	`<gml:coordinates>`
	`(...)`
	`</gml:coordinates>`
	`</gml:LinearRing>`
	`</gml:outerBoundaryIs>`
	`</gml:Polygon>`
	`</gml:polygonMember>`
	`</gml:MultiPolygon>`
	`</GML></ENTITY>`

Table 1: Summary of processing for "the streets between Boston Univ and Boston Common".

5.1.2 Areal Entities Derived from Linear Entities

The previous example considered the problem of deriving linear entities from areal entities; this example considers the opposite problem. The goal now is to find a set of areal entities. The input phrase for this example is "the school on Commonwealth Ave". Because this example is very similar to the previous one, it is useful to discuss in depth only the differences in the processing done by the Grounder module. The other steps follow almost the same procedures as in the previous example, except that there is only one named entity and therefore there is no conjunction.

Tagger: The input phrase is tagged, giving "the school on Commonwealth_Ave/LOCATION".

Parser: The parse tree for the tagged phrase is shown below in Figure 14.

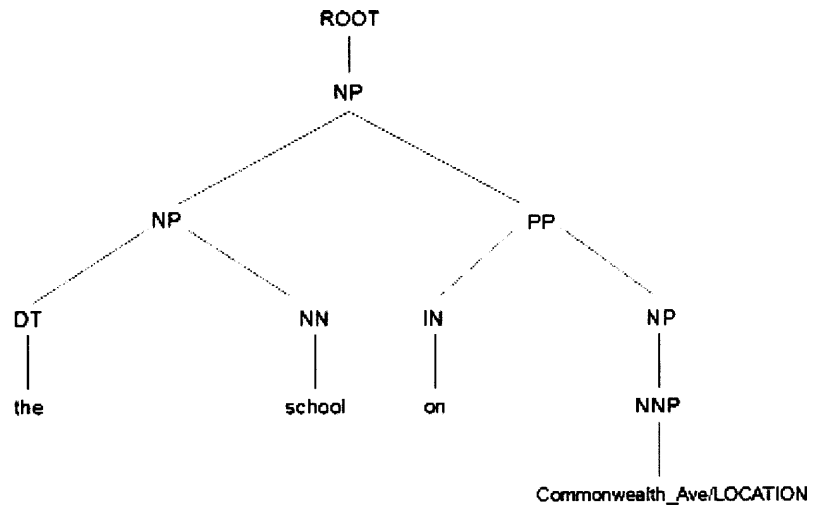

Figure 14: Parse tree for "the school on Commonwealth_Ave/LOCATION".

Reasoner: Because there is only one named entity and no conjunction, the only *NamedFeature* is "(NNP Commonwealth Ave/LOCATION)". "(NP (NNP Commonwealth_Ave/LOCATION))" gets asserted as a *FeatureSet* with "*hasFeature* (NNP Commonwealth_Ave/LOCATION)". Similarly, "(NP (DT the) (NN school))" gets asserted as a *FeatureSet* with "*hasFeature* (NN school)". The relation "(NN school) *on* (NP (NNP Commonwealth_Ave/LOCATION))" is then asserted, followed by the *intersects-buffer-convexHull* property. The resulting query is similar to that used in the previous example.

Grounder: The assertions made by the Reasoner module are now used by the Grounder module, resulting in a set of 480 *MultiLineString*s for "Commonwealth Ave" and 1 *MultiPolygon*

for "school".

Displayer: The set of geometries from the Grounder module is converted into a KML string for display in Google Earth. The resulting display is shown below in Figure 15.

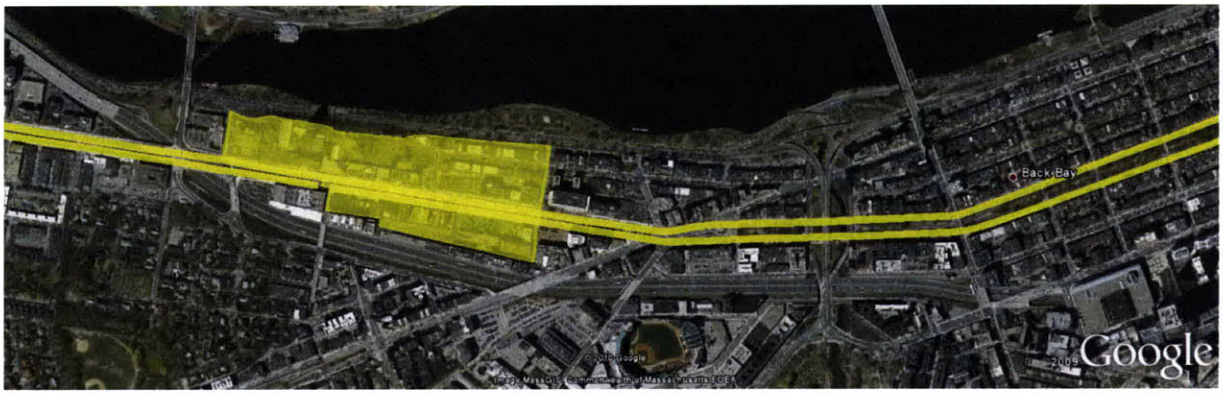

Figure 15: Result set of geometries for "the school on Commonwealth Ave".
(Display credit: Google Earth.)

In this example, the only result for "the school on Commonwealth Ave" was the geometry corresponding to Boston University. However, Boston College is another university that is on Commonwealth Ave. It does not appear as a result in this example simply because it was not in the shapefiles that comprised our research group's TIGER database. (Our database was limited to the major features within the state of Massachusetts and some locations within Boston.) GeoNames cannot be used to find *UnnamedFeature*s because, at the time of this writing, there is no practical way to search through it in the same way PostGIS is used to search through TIGER.

Of course, GeoCoder can only find what it has available to it, so if GeoCoder cannot search through its databases for that location or the location is missing altogether from the databases, then that location will not appear as a result. Furthermore, there are in fact many streets spread throughout Massachusetts that are named "Commonwealth Ave". This example does not show these additional segments because there are no schools on them that our database mentions.

Table 2 gives a summary of the processing results in this example.

Module	Result
Tagger	"the school on Commonwealth_Ave/LOCATION"
Parser	(ROOT (NP (NP (DT the) (NN school)) (PP (IN on) (NP (NNP Commonwealth_Ave/LOCATION))))))
Reasoner	(NP (NNP Commonwealth_Ave/LOCATION)) *rdf*:type *FeatureSet*; (NP (NNP Commonwealth_Ave/LOCATION)) *hasFeature* (NNP Commonwealth_Ave/LOCATION); (NP (DT the) (NN school)) *rdf*:type *FeatureSet*; (NP (DT the) (NN school)) *hasFeature* (NN school); (NN school) *on* (NP (NNP Commonwealth_Ave/LOCATION)); (NN school) *intersect-buffer-convexHull* (NP (NNP Commonwealth_Ave/LOCATION)); (NN school) *rdf*:type *ExistingFeature*;
Grounder	480 *MultiLineString*s for "Commonwealth Ave" 1 *MultiPolygon* for "school"
Displayer	<ENTITY><NAME>Commonwealth Ave</NAME><GML> <gml:MultiLineString srsName='0'> <gml:lineStringMember> <gml:LineString srsName='0'> <gml:coordinates> *(...)* </gml:coordinates> </gml:LineString> </gml:lineStringMember>

```
</gml:MultiLineString>
</GML></ENTITY>

(one ENTITY for each street segment)

<ENTITY><NAME>school</NAME><GML>
 <gml:MultiPolygon srsName='0'>
  <gml:polygonMember>
    <gml:Polygon srsName='0'>
      <gml:outerBoundaryIs>
        <gml:LinearRing srsName='0'>
          <gml:coordinates>
            (...)
          </gml:coordinates>
        </gml:LinearRing>
      </gml:outerBoundaryIs>
    </gml:Polygon>
  </gml:polygonMember>
</gml:MultiPolygon>
</GML></ENTITY>
```

Table 2: Summary of processing for "the school on Commonwealth Ave".

5.2 Path Resolution

The following sections describe the process of path resolution, using an example with a path containing two vertices in Section 5.2.1 and an example with a path containing three vertices in Section 5.2.2.

5.2.1 2-Vertex Path Resolution

This example focuses on the problem of finding a path given two named locations as the path vertices. The input phrase is “the path from Boston Univ to Boston Common”, which is in

the canonical form for a path expression and uses the same two locations used earlier, "Boston Univ" and "Boston Common". The goal here is to obtain a set of geometries that represent the shortest path (using streets) from Boston University to Boston Common. GeoCoder will detect that the input phrase is in the canonical form for a path expression and will follow the right-side branch in Figure 5, specific to pathfinding. As explained in Section 2.3, path resolution involves three steps: calling the Resolver module to resolve each path vertex, calling the Resolver module to resolve all possible connecting geometries, and calling the Pathfinder module on the results. The resulting path is then passed to the Displayer to present the results.

Resolver: The Resolver module is called to obtain a geometry for the path's start vertex, "Boston Univ". The steps taken by the Resolver module are a straightforward chain of processing through the Tagger, Parser, Reasoner, and Grounder (as in Section 5.1.1).

Resolver: The Resolver module is called to obtain a geometry for the path's end vertex, "Boston Common".

Resolver: The Resolver module is called to obtain all intermediate, connecting geometries between the start and end path vertices. The input phrase to the Resolver is "the streets between Boston Univ and Boston Common", and the result is the set of geometries representing streets that intersect the convex hull formed from the geometries for Boston University and Boston

Common (as in Section 5.1.1).

Pathfinder: Together, all of the geometries obtained in the previous three steps form the graph of geometries. This graph is the same as the set of geometries shown in Figures 11, 12, and 13. The Pathfinder module is called on the graph, with the two path vertices designated as the endpoints of the path. As stated before, the Pathfinder uses a modified version of Dijkstra's algorithm that acts on geometries. The result is a set of geometries that includes the path vertices and the shortest path between them.

Displayer: The set of geometries from the Pathfinder is passed to the Displayer, which produces output as shown below in Figure 16.

Figure 16: The path resulting from the input phrase "the path from Boston Univ to Boston Common". *(Display credit: Google Earth.)*

Table 3 gives a summary of the processing results in this example.

Module	Result
Resolver	1 *MultiPolygon* for "Boston Univ"
Resolver	1 *MultiPolygon* for "Boston Common"
Resolver	1224 *MultiLineString*s for "streets"
Pathfinder	Path from "Boston Univ" to "Boston Common"
Displayer	<ENTITY> <NAME>Boston Univ</NAME> <GML><gml:MultiPolygon srsName='0'> <gml:polygonMember> <gml:Polygon srsName='0'> <gml:outerBoundaryIs> <gml:LinearRing srsName='0'> <gml:coordinates> *(coordinates omitted)* </gml:coordinates> </gml:LinearRing> </gml:outerBoundaryIs> </gml:Polygon> </gml:polygonMember> </gml:MultiPolygon> </GML></ENTITY> <ENTITY> <NAME>Boston Common</NAME> <GML><gml:MultiPolygon srsName='0'> <gml:polygonMember> <gml:Polygon srsName='0'> <gml:outerBoundaryIs> <gml:LinearRing srsName='0'> <gml:coordinates> *(coordinates omitted)* </gml:coordinates> </gml:LinearRing> </gml:outerBoundaryIs> </gml:Polygon> </gml:polygonMember> </gml:MultiPolygon> </GML></ENTITY> <ENTITY> <NAME>street</NAME> <GML><gml:MultiLineString srsName='0'> <gml:lineStringMember> <gml:LineString srsName='0'> <gml:coordinates> *(coordinates omitted)* </gml:coordinates>

	</gml:LineString> </gml:lineStringMember> </gml:MultiLineString> </GML></ENTITY>

Table 3: Summary of processing for "the path from Boston Univ to Boston Common".

5.2.2 3-Vertex Path Resolution

It is straightforward to accommodate more than two vertices for a path. The entire process just described finds the path between any two path vertices that are adjacent in sequence, so repeating this process for every pair of sequentially-adjacent vertices will give all of the subpaths. The final result may be found by simply taking the union of all of the subpaths.

As an example, the phrase "the path from Fenway Park to Boston Univ to Boston Common" adds another path vertex to the front of the path. GeoCoder would first find the path from the vertex for "Fenway Park" to the vertex for "Boston Univ", then find the path from "Boston Univ" to "Boston Common", and then take the union of the two results. The result is shown below in Figure 17.

Figure 17: The path resulting from the input phrase "the path from Fenway Park to Boston Univ to Boston Common". *(Display credit: Google Earth.)*

5.3 Nongeospatial Phrase Resolution Using START

The following sections describe the process of nongeospatial phrase resolution using START. Section 5.3.1 covers the process for a basic phrase and Section 5.3.2 discusses the process when the same phrase is used as part of a larger phrase.

5.3.1 Simple Nongeospatial Phrase Resolution

This section discusses the resolution of the nongeospatial phrase "the most populous city in Massachusetts", with Section 5.3.1.1 using the singular *UnnamedFeature* "city" and Section

5.3.1.2 using the plural *UnnamedFeature* "cities".

5.3.1.1 Singular *UnnamedFeature*

This example follows the resolution of a simple nongeospatial phrase, using START. Now one path through the Arbiter will be explained in detail, as it is the Arbiter that allows this problem to be solved. As mentioned before, the Arbiter attempts to resolve a nongeospatial phrase to a geospatial one by partitioning the phrase. The input phrase for this example is "the most populous city in Massachusetts". While this phrase is describing a location with the geospatial property that it is in Massachusetts, the reliance on the nongeospatial property of population causes the phrase to be outside of GeoCoder's native scope. The goal is to resolve this phrase to its geospatial equivalent, "Boston". Then the phrase can be resolved to a geometry through the steps described previously. This example therefore focuses on explaining the actions of the Arbiter module.

Tagger: The input phrase is passed to Tagger module and becomes "the most populous city in Massachusetts/LOCATION".

Parser: The tagged phrase is passed to the Parser. The resulting parse tree is the same as that in Figure 3.

Arbiter, Phrase Tree Generation: The parse tree is passed to the Arbiter, which produces the same recursive phrase tree as that in Figure 4.

Arbiter, Phrase Partitioning, Step 1: Phrase partitioning begins. The final goal for phrase partitioning is to obtain as large a nongeospatial phrase as possible so that it can be passed to START, while determining whether the current subphrase is geospatial so that it can be resolved by the Resolver module. The Arbiter selects the rightmost terminal node, "Massachusetts", to be the first subphrase. The state of the phrase partitioning is shown below in Figure 18, where the box around "Massachusetts" indicates that it is the current subphrase.

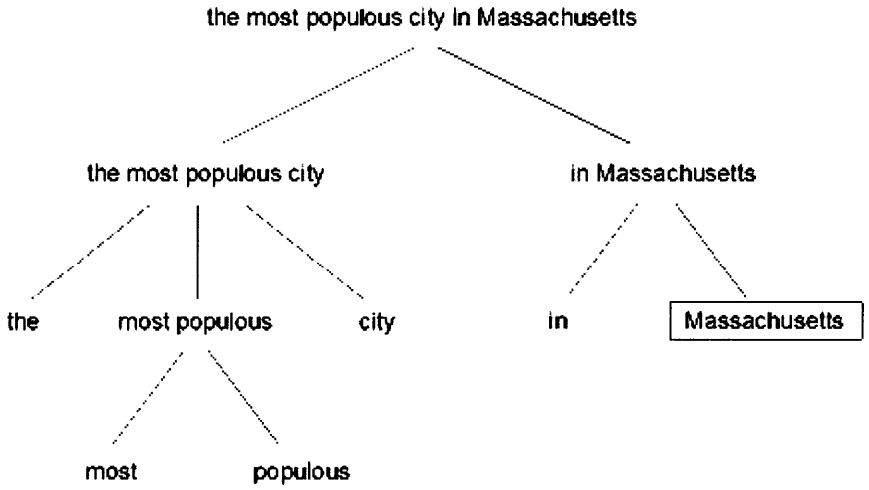

Figure 18: State of the phrase partitioning at Step 1.

The Resolver module processes this subphrase through the Tagger, Parser, Reasoner, and Grounder modules. The Grounder looks for this subphrase in TIGER, but cannot find Massachusetts there, so it then asks GeoNames for coordinates, of which GeoNames returns two points:

```
 42.365647002811897, -71.108322143554702
-18.216666700000001, 29.866666666666667
```

Because there are two points corresponding to "Massachusetts", both must be considered. Each point is used to create a circular areal geometry with that point as its center. Of course, this does not end up being useful for this example, as this areal geometry does not accurately represent Massachusetts. However, the purpose of this step is to determine whether this subphrase is geospatial or nongeospatial in nature. Because GeoNames has points for this subphrase, it is assumed that the subphrase is geospatial. If it were nongeospatial, GeoNames would have no entry for it, and then START would be queried.

The call to GeoNames resulted in the creation of areal geometries, so the current node is considered "successful".

Arbiter, Phrase Partitioning, Step 2: Because the last phrase node was successful, its next-left sibling would have been selected by the Arbiter, and "in" would have been set as the current subphrase. However, because there is only one sibling left, the algorithm skips this node

and moves up to the parent.

Arbiter, Phrase Partitioning, Step 3: The parent is now the current node and "in Massachusetts" is now the current subphrase. The state of the phrase partitioning is shown below in Figure 19. The struck-out phrase "in" and the thickened, struck-out branch indicate that the corresponding node was skipped, which, for the purposes of the algorithm, is equivalent to failure. The underlined phrase "Massachusetts" and the thickened branch indicate that the corresponding node was successful.

The Resolver now processes "in Massachusetts" and detects "Massachusetts" as a named feature, so it attempts to find it in TIGER, but it does not find a result, as before. GeoNames is then asked for the coordinates of Massachusetts and areal geometries are again produced. This causes this node to be marked as successful.

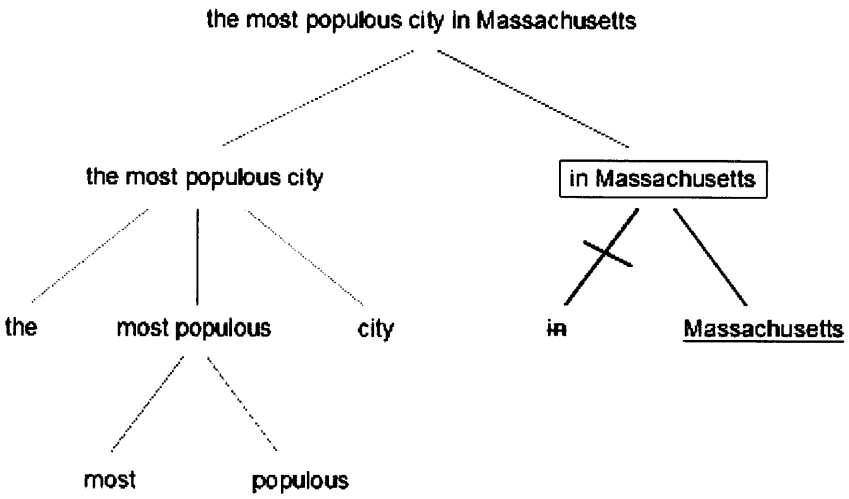

Figure 19: State of the phrase partitioning at Step 3.

Arbiter, Phrase Partitioning, Step 4: The node labeled "in Massachusetts" succeeded, so the next node to check would have been its next-left sibling, the node labeled "the most populous city", but because this is the only sibling, the algorithm skips it and moves up to the parent, "the most populous city in Massachusetts". However, if it had not been the only sibling, then the algorithm would have moved to its rightmost child, the "city" node. This child node has no children so this would have meant the algorithm had moved to the rightmost unvisited terminal node, and the algorithm would have continued as normal. Ultimately, though, the results of the entire subtree rooted at "the most populous city" would not have mattered, so this subtree is skipped and the algorithm moves up to the parent node, the root.

Arbiter, Phrase Partitioning, Step 5: The root is now the current node and "the most

populous city in Massachusetts" is now the current subphrase. The state of the phrase partitioning is shown below in Figure 20.

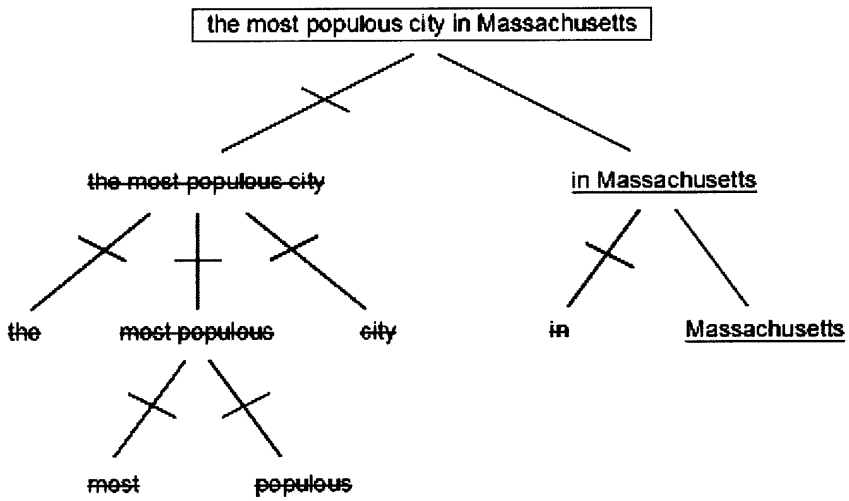

Figure 20: State of the phrase partitioning at Step 5.

This subphrase does not yield a result from TIGER or GeoNames, but the Arbiter queries START, which returns the following list:

Boston, Massachusetts
Worcester, Massachusetts
Springfield, Massachusetts
Lowell, Massachusetts
Cambridge, Massachusetts
Brockton, Massachusetts
New Bedford, Massachusetts
Fall River, Massachusetts
Lynn, Massachusetts
Quincy, Massachusetts

By the specification of the interface to START developed for GeoCoder, this list is sorted with the most populous city at the top, and since the phrase asks for just the one most populous city, the desired answer is "Boston, Massachusetts". Because TIGER and GeoNames work with names and not name-state combinations, it is necessary to extract just "Boston" from this result, and so the original nongeospatial phrase "the most populous city in Massachusetts" is replaced with "Boston". This new phrase is then passed through the system one more time.

Tagger: The new phrase "Boston" is passed to the Tagger, which returns "Boston/LOCATION".

Parser: The tagged phrase is passed to the Parser module, which returns the simple parse tree shown below in Figure 21.

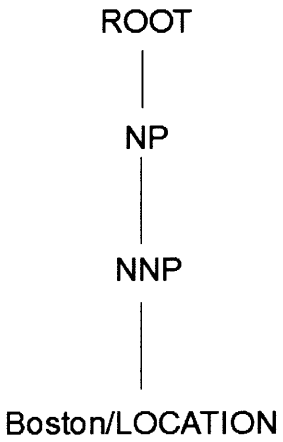

Figure 21: Parse tree for "Boston".

Reasoner: The parse tree is passed to the Reasoner, which asserts the "(NNP Boston/LOCATION)" node as a *NamedFeature.* This causes the "(NP (NNP Boston/LOCATION))" to be asserted as a *FeatureSet.*

Grounder: The Grounder resolves the "(NP (NNP Boston/LOCATION))" *FeatureSet* to a geometry.

Displayer: Finally, the Displayer presents the geometry for Boston, as shown below in Figure 22.

Figure 22: The resulting geometry for the nongeospatial phrase "the most populous city in Massachusetts". *(Display credit: Google Earth.)*

Table 4 gives a summary of the processing results in this example.

Module	Result
Tagger	"the most populous city in Massachusetts/LOCATION"
Parser	`(ROOT (FRAG (NP (DT the) (ADJP (RBS most) (JJ populous)) (NN city)) (PP (IN in) (NP (NNP Massachusetts/LOCATION)))))`
Arbiter (Phrase Tree Generation)	`(the most populous city in Massachusetts (the most populous city the (most populous most populous) city) (in Massachusetts in Massachusetts))`
Arbiter (Phrase Partitioning)	"Boston"
Tagger	"Boston/LOCATION"
Parser	`(ROOT (NP (NNP Boston/LOCATION)))`
Reasoner	(NP (NNP Boston/LOCATION)) *rdf*:type *FeatureSet*; (NP (NNP Boston/LOCATION)) *hasFeature* (NNP Boston/LOCATION);
Grounder	1 *MultiPolygon* for "Boston"
Displayer	`<ENTITY><NAME>Boston</NAME><GML> <gml:MultiPolygon srsName='0'> <gml:polygonMember> <gml:Polygon srsName='0'> <gml:outerBoundaryIs> <gml:LinearRing srsName='0'> <gml:coordinates> -71.189597,42.281167 -71.18974,42.281236 -71.190044,42.281415` *(remaining coordinates omitted)* `</gml:coordinates>`

	</gml:LinearRing> </gml:outerBoundaryIs> </gml:Polygon> </gml:polygonMember> </gml:MultiPolygon> </GML></ENTITY>

Table 4: Summary of processing for "the most populous city in Massachusetts".

5.3.1.2 Plural *UnnamedFeature*

In the previous example, the input phrase "the most populous city in Massachusetts" asked for the most populous *city,* but it is possible to check for the plural forms of the locations to be derived, in order to return multiple results. In this case, inputting the phrase "the most populous cities in Massachusetts" will cause GeoCoder to accept the entire list returned from START as results. Each individual result will then be processed through the Resolver and Displayer modules. The result is shown below in Figure 23.

Currently, GeoCoder is programmed to detect the plural forms of certain common location types, such as *city* and *park*, but potentially the plural form of any word may be detected by using the WordNet lexical database [21]. The equivalent singular form of a word may be looked up in the database and if that word corresponds to a location type (such as *city*), then GeoCoder can be instructed to accept multiple results of that type.

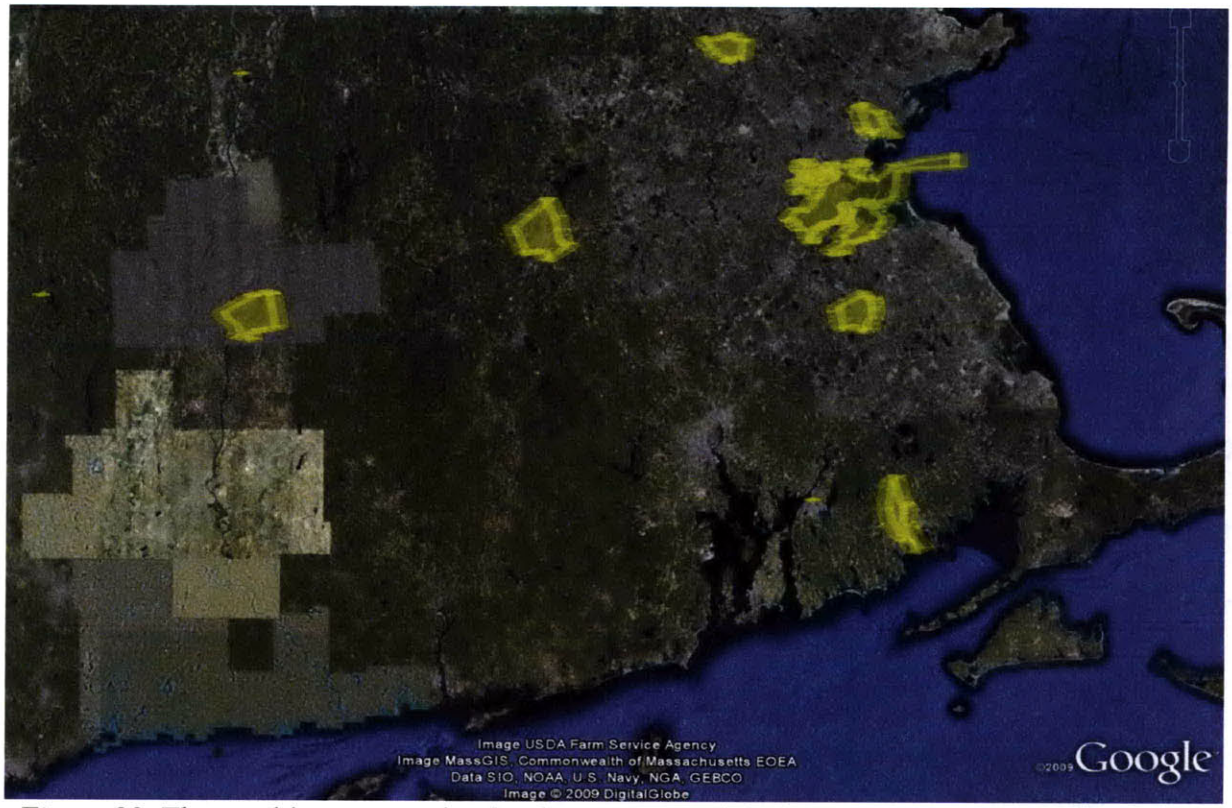

Figure 23: The resulting geometries for the nongeospatial phrase "the most populous cities in Massachusetts". *(Display credit: Google Earth.)*

5.3.2 Nongeospatial Subphrase in a Larger Phrase

In the examples described in Sections 5.3.1.1 and 5.3.1.2, it may have seemed redundant to go through phrase partitioning only to have the entire phrase passed to START in the end. Indeed, in some cases it is redundant, but the purpose of phrase partitioning is for GeoCoder to attempt to discover and resolve nongeospatial subphrases without knowing beforehand what part

of the input phrase is nongeospatial. The previous two examples served to explain nongeospatial phrase resolution on its own without complications; now a small but important extension to the previous input phrase is made and the value of phrase partitioning should become clear.

Suppose the input phrase is now "the cities beside the most populous city in Massachusetts". This input phrase is significantly more demanding than the previous two, as now the phrase contains a relation between a geospatial subphrase and a nongeospatial subphrase. Without phrase partitioning, GeoCoder would not know that only the subphrase "the most populous city in Massachusetts" is nongeospatial. GeoCoder would attempt to resolve the entire phrase and fail; then pass the phrase to GeoNames, which would fail; and finally pass the phrase to START, which would also fail. However, with phrase partitioning, the process described in Section 5.3.1.1 would resolve "the most populous city in Massachusetts" to "Boston" and the input phrase would become "the cities beside Boston", which is certainly resolvable by GeoCoder, using TIGER alone. The initial phrase tree is shown in Figure 24 and the phrase tree after resolution of "the most populous city in Massachusetts" to "Boston" is shown in Figure 25. The final result is shown in Figure 26. (TIGER designates some areas of water as belonging to various cities.)

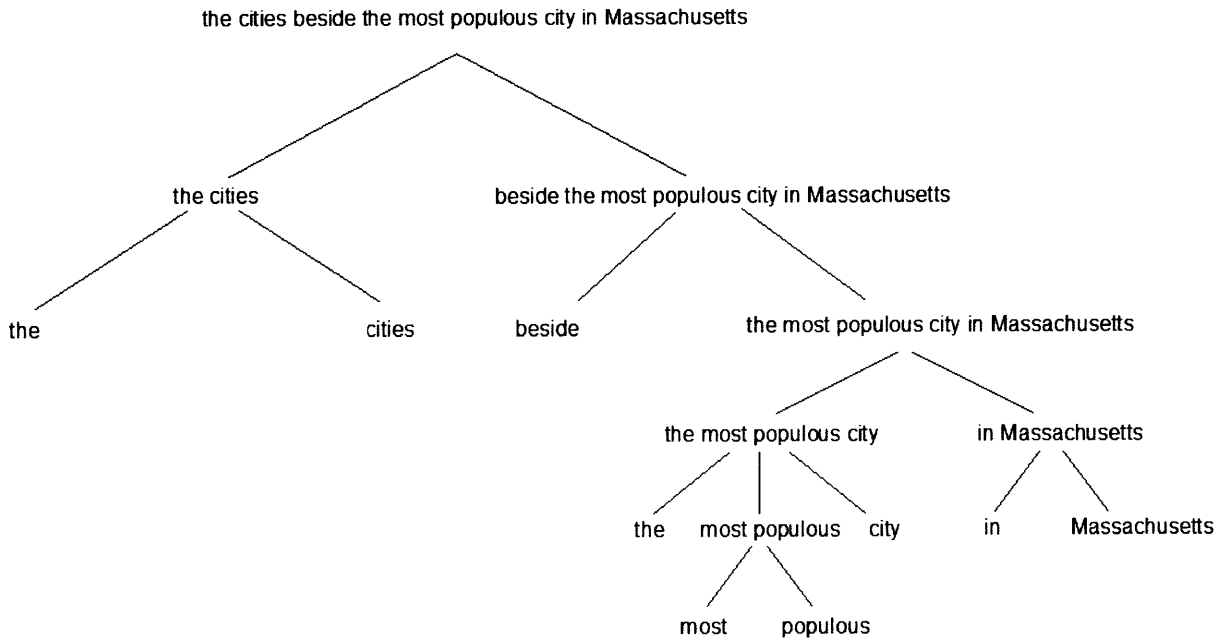

Figure 24: Phrase tree for "the cities beside the most populous city in Massachusetts" at the start of the phrase partitioning algorithm.

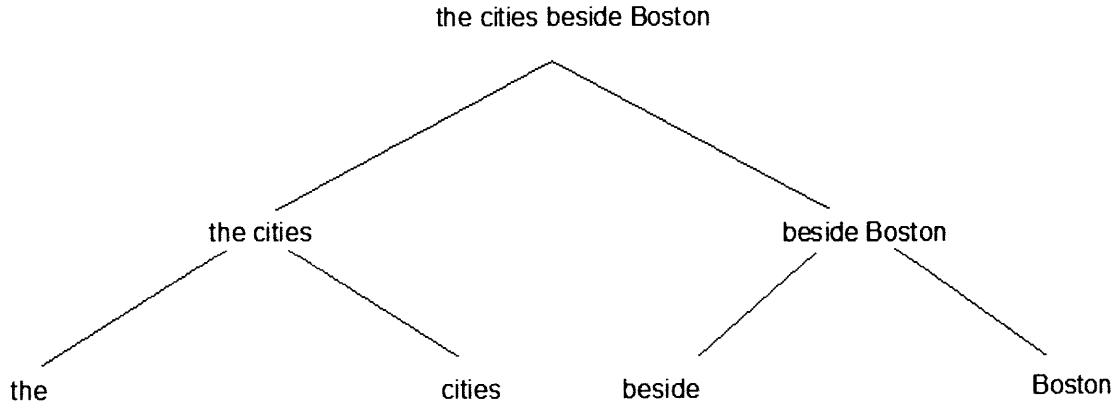

Figure 25: Phrase tree for "the cities beside the most populous city in Massachusetts" after resolution of "the most populous city in Massachusetts" to "Boston".

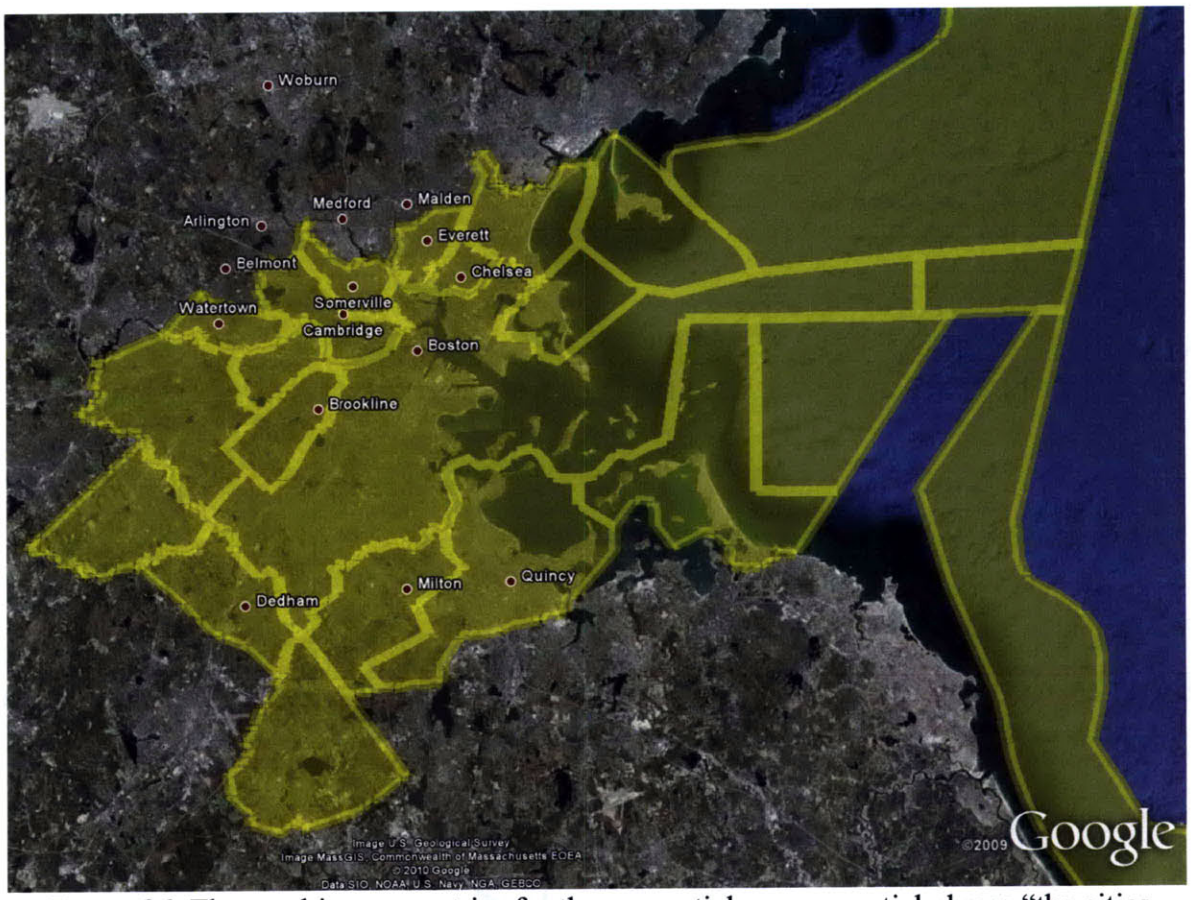

Figure 26: The resulting geometries for the geospatial-nongeospatial phrase "the cities beside the most populous city in Massachusetts". *(Display credit: Google Earth.)*

Chapter 6

Future Work

6.1 Improved Phrase Partitioning and Delegation

As described, the Arbiter module uses phrase partitioning to determine what phrases to send to START. The purpose of this process is to resolve nongeospatial phrases (as opposed to geospatial ones, which GeoCoder can resolve using TIGER or GeoNames). However, it is possible for this process to attempt to resolve subphrases that on their own make no sense. For example, for the phrase, "the birthplace of John Adams", phrase partitioning would dictate that "Adams" be resolved, which would actually lead to results (for example, "Adams" is a city in Massachusetts"). If phrase partitioning continues and designates a larger phrase that includes this subphrase, then this misstep is inconsequential, but if there had been a situation where a subphrase of this nature had been passed to START, then START may also have returned a nongeospatial answer. As explained before, the resolution of nongeospatial phrases must return geospatial phrases or the resulting complete phrase (when the new answer replaces the old subphrase) will be incomprehensible to GeoCoder. Currently, phrase partitioning attempts to

resolve all subphrases it generates. Improving the process of phrase partitioning and better determining when it is best to attempt to resolve a subphrase would help avoid the problems mentioned above.

6.2 Further Integration with START

GeoCoder can interface with START and ask it questions via the interface developed for this research. START returns some information it has extracted from the input phrase, such as what kind of object is being asked for. Currently, GeoCoder only uses a small amount of this information. Because phrase partitioning is used to pass phrases to START about which GeoCoder can discern nothing, the information returned from START could assist in any subsequent reasoning that is added in future work. For example, if START indicates that the phrase mentions a list of rivers, then GeoCoder could determine that the phrase contains bodies of water. This would be relevant for any further reasoning that distinguishes between land, water, and air transportation channels, for example.

6.3 Less Reliance on Canonical Forms, Varied Forms

Section 3.3 discusses two canonical forms for input phrases to GeoCoder. These two forms restrict the structure of phrases the user can input. However, actual language is much more

varied than just these forms. It would be more natural to allow for variations in these phrase forms that capture a wider variety of language. Of course, it would be even better to rely less on canonical forms, but this is a much more difficult endeavor.

6.4 Verbs and Modality

Currently, GeoCoder does not allow for verbs in input phrases. GeoCoder cannot detect these and has no processing in place that can handle them. As mentioned in Section 6.2, one possible direction is to improve GeoCoder's reasoning capabilities in the area of modality. Specifically, GeoCoder could be improved to distinguish between land, water, and air locations and modalities of travel, use the input phrase to determine what modality of travel is being used (perhaps through the use of verbs, such as "walk" or "swim"), and use this information in its grounding and resolution processes. This functionality may be integrated with the pathfinding functions to describe new kinds of paths, such as those that travel through water channels or those that change modalities midway (for example, "the path from MIT to the Charles River to Boston Harbor").

6.5 Improved Distance Metric for Scale

Section 4.4 describes the distance metric used to determine whether two locations are "near" each other. This metric adds the areas of the two objects and multiplies them by a scale factor. While this metric grants GeoCoder a reasonable definition of nearness, it likely can be improved. For example, linear geometries have length, but they do not have area, so it is impossible to determine whether another location is near a linear geometry, unless the location is actually *on* the linear geometry. One possible solution is to use a metric for linear geometries that is different than that used for areal geometries, such as one that uses length instead of area. This would need to account for all possible combinations of comparisons between linear and areal geometries, however. Another solution is to use other aspects that are common to both linear and areal geometries, such as the length of the defining lines (for linear geometries, this is just the length, but for areal geometries, this is the length of the border).

Chapter 7

Conclusion

This thesis builds upon Slagle's original work by addressing some of its limitations and extending the concepts presented therein. The main contributions are the following:

- GeoCoder can now handle and reason upon linear geometries in order to represent linear entities, such as borderlines, boundaries, streets, and so on. This allows a solution to the problem of path resolution.
- GeoCoder can now detect and resolve path expressions that describe a path consisting of both areal and linear entities. The process uses a modified form of Dijkstra's algorithm developed for this thesis, which acts on geometries rather than on points.
- GeoCoder can now use phrase partitioning to select nongeospatial phrases, which it can then attempt to resolve through interaction with the START natural language question answering system.
- GeoCoder is now highly modularized, allowing for greater control over its functions. The specific problem now dictates what aspects of GeoCoder used and when. This modularization is what allows solutions to the problems of path resolution and

nongeospatial phrase resolution.

- GeoCoder now uses a dynamic distance metric to determine whether two locations are "near" one another. This metric uses the areas of the locations in question, so that the problem of determining whether two cities are near each other uses a different distance threshold than the problem of determining whether two buildings are near each other.
- GeoCoder now has greatly improved performance due to use of a dynamic ruleset. Upon receiving an input phrase, GeoCoder loads only the rules relevant to the prepositions detected in the phrase.

Appendix A.1

Slagle's Geospatial Phrase Resolution Algorithm

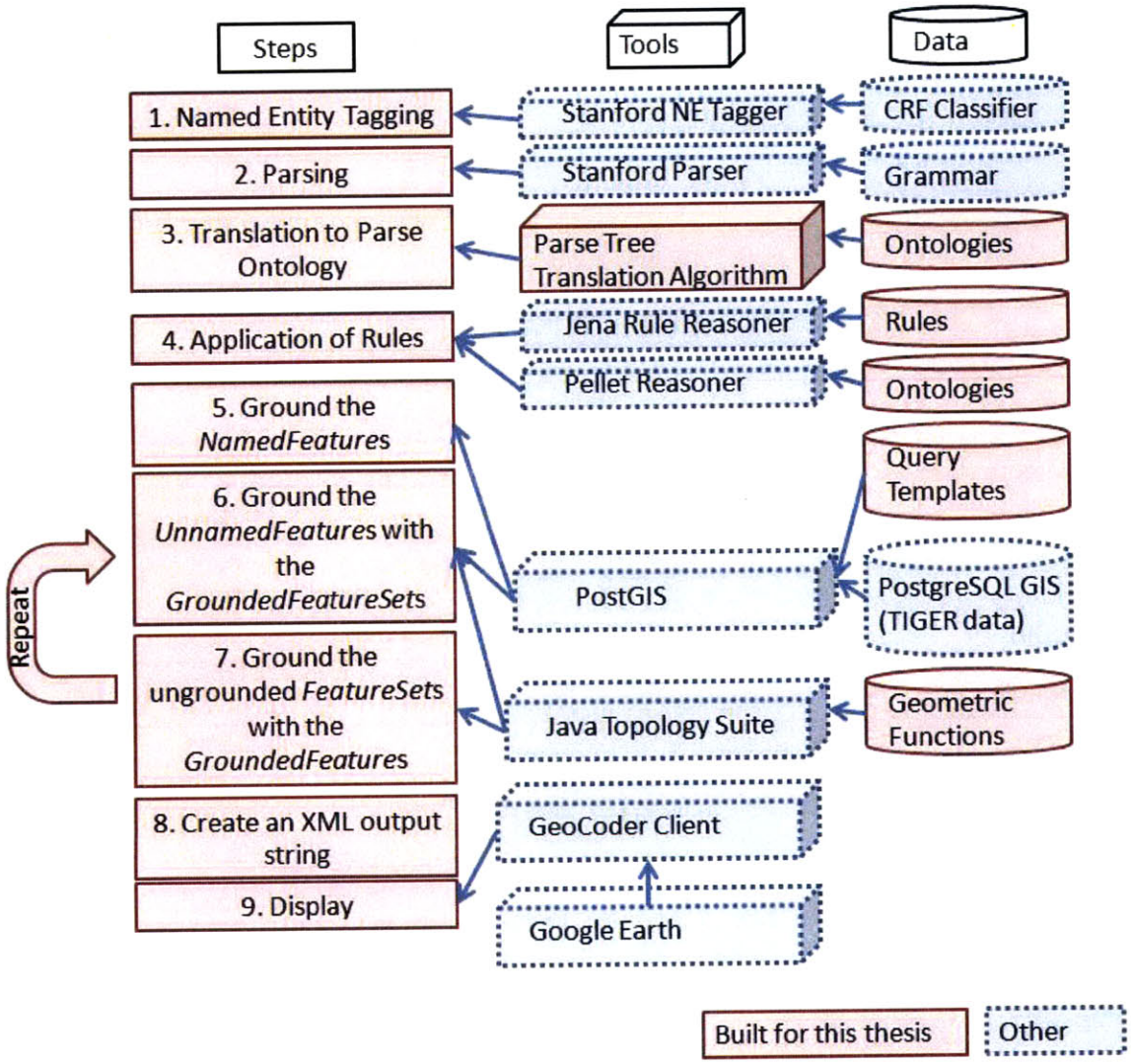

Figure 27: Outline of the process taken by Slagle's GeoCoder system to resolve a given input phrase to a geometry. *(Image credit: Slagle [3].)*

Appendix A.2

Geometric Pathfinding Algorithm (Modified Dijkstra's Algorithm)

```
create intersection table (maps two geometries to their coordinate of
intersection)

// build the graph
for all geometries:
      if two geometries intersect:
            mark them as neighbors (add them to intersection tables)
            find the centroid of their intersection and record it

for each coordinate c in the intersection table (the values):
      set the distance of c from the start geometry to infinity

// pathfind
create queue
add all the intersection points to the queue
while queue is not empty:
      set u to the next coordinate in the queue
      set smallestDist to the distance from the start geometry

      for each coordinate c in the queue:
            set currDist to the distance of c from the start geometry
            if currDist < smallestDist:
                  set u to c
```

```
                set smallestDist to currDist

      if u is at infinite distance:
            break
      remove u from the queue
      if u is the endpoint:
            break

      Get the geometries using u as an intersection point.
      For each of these geometries, get their intersection points
      (these are the neighbors to the original coordinate, u).
      for each intersection point:
            determine the distance from the start to this point
            if this is smaller than the current value:
                  update the value
                  add u as its previous point

// Build the final path
create a list of geometries, the final path
set coordinate u to the endpoint
set coordinate lastu to null
while u's previous point is not null:
      find u's previous point
      add the point to the final path
      set this previous point as u
return the final path
```

Appendix A.3

Description of TIGER

The various tables in the TIGER database are shown below in Figure 28. GeoCoder, when searching through TIGER for a geometry corresponding to a location, looks in particular tables depending on the nature of the location. If the location is an areal entity and on land, GeoCoder searches in *arealm*. If the location is an areal entity and is a body of water, GeoCoder searches in *areawater*. Finally, if the location is a linear entity, whether on land or water, GeoCoder searches in *edges*. It is important to note that linear entities are represented in TIGER by *multiple* segments, not just one. Therefore, a street such as "Commonwealth Ave" has multiple geometries in the *edges* table that must all be collected in order to completely represent the street.

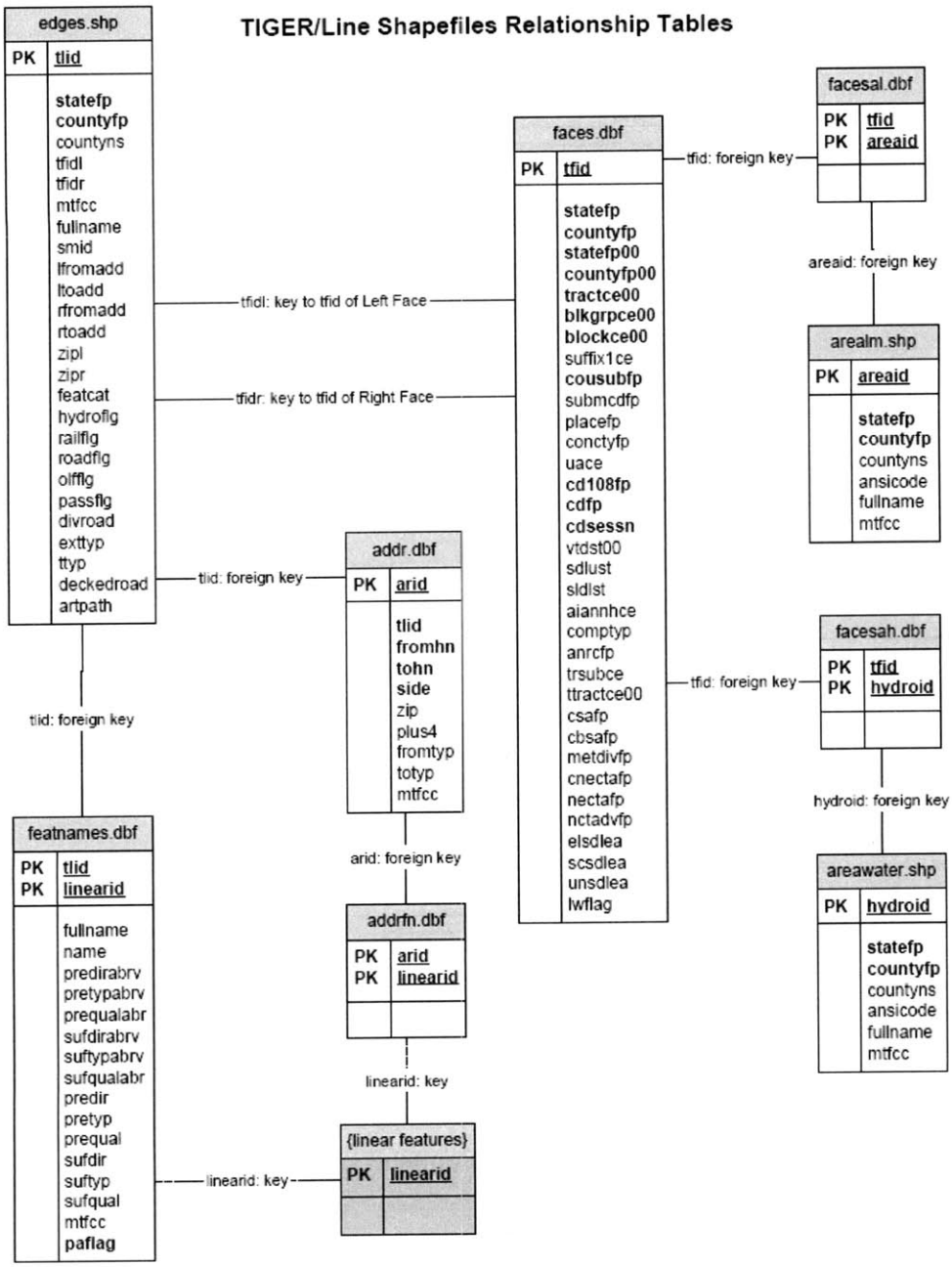

Figure 28: The TIGER relationship tables. *(Image credit: U. S. Census Bureau Geography Division [20].)*

References

[1] Google, "Google Maps", URL: http://maps.google.com [accessed September 28, 2009].

[2] "MapQuest", URL: http://www.mapquest.com [accessed September 28, 2009].

[3] Slagle, Amy Michelle, "Geospatial Phrase Grounding and Disambiguation".

[4] Stanford Natural Language Processing Group, "Named Entity Recognition (NER) and Information Extraction (IE)", URL: http://nlp.stanford.edu/ner/ [accessed January 24, 2010].

[5] Stanford Natural Language Processing Group, "The Stanford Parser: A statistical parser", URL: http://nlp.stanford.edu/software/lex-parser.shtml [accessed January 24, 2010].

[6] Dickinson, I., "The Jena Ontology API", URL: http://jena.sourceforge.net/ontology/index.html [accessed January 24, 2010].

[7] Bikel, D. M., Miller, S., Schwartz, R., and Weischedel, R., "Nymble: A High-Performance Learning Name-Finder," Proceedings of the Fifth Conference on Applied Natural Language Processing, Morgan Kaufman Publishers, San Francisco, April 1997, pp. 194-201.

[8] Zhou, G. and Su, J., "Named Entity Recognition Using an HMM-Based Chunk Tagger," Proceedings of the 40th Annual Meeting of the Association for Computational

Linguistics, Association for Computational Linguistics, Morristown, NJ, July 2002, pp. 473-480.

[9] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y., "Pellet: A Practical OWL-DL Reasoner", *Journal of Web Semantics*, Vol. 5, June 2007, pp. 51-53.

[10] "PostGIS: Home", URL: http://www.postgis.org [accessed January 24, 2010].

[11] Vivid Solutions, Inc., "JTS Topology Suite", URL: http://www.vividsolutions.com/jts/jtshome.htm [accessed January 24, 2010].

[12] U. S. Census Bureau Geography Division, "U. S. Census Bureau – TIGER/Line®", URL: http://www.census.gov/geo/www/tiger/ [accessed January 24, 2010].

[13] Boris Katz, Gary Borchardt and Sue Felshin, "Natural Language Annotations for Question Answering", Proceedings of the 19th International FLAIRS Conference (FLAIRS 2006), May 2006, Melbourne Beach, FL. URL: http://start.csail.mit.edu/ [accessed January 24, 2010]

[14] Google, "Google Earth", URL: http://earth.google.com [accessed January 24, 2010].

[15] "About GeoNames", URL: http://www.geonames.org/about.html [accessed January 24, 2010].

[16] Leidner, J. L., "Toponym Resolution: A First Large-Scale Comparative Evaluation," School of Informatics, University of Edinburgh, Informatics Research Report, July 2006.

[17] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and

Cybernetics SSC4, Vol. 4, July 1968, pp. 100–107.

[18] Dijkstra, E. W., "A note on two problems in connexion with graphs". Numerische Mathematik 1, 1959, pp. 269–271.

[19] Waldinger, R., Jarvis, P., and Dungan, J., "Using Deduction to Choreograph Multiple Data Sources," Semantic Web Technologies for Searching and Retrieving Scientific Data, Sanibel Island, FL, October 2003.

[20] U. S. Census Bureau Geography Division, "Description of the Relationship Tables", URL: http://www.census.gov/geo/www/tiger/rel_file_desc.pdf [accessed September 28, 2009].

[21] George A. Miller (1995). "WordNet: A Lexical Database for English." Communications of the ACM Vol. 38, No. 11: 39-41.